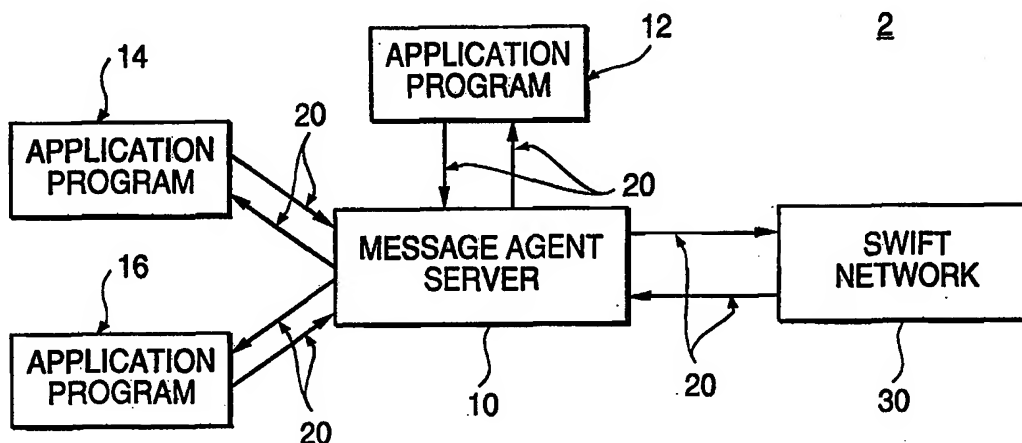




INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

| | | |
|--|-----------|--|
| (51) International Patent Classification ⁶ : H01J 13/00 | A1 | (11) International Publication Number: WO 98/56024 (43) International Publication Date: 10 December 1998 (10.12.98) |
| (21) International Application Number: PCT/US98/10930 (22) International Filing Date: 5 June 1998 (05.06.98) (30) Priority Data: 60/050,422 5 June 1997 (05.06.97) US (71) Applicant: CROSSMAR, INC. [US/US]; 111 Wall Street, New York, NY 10005 (US). (72) Inventors: JACOBS, David, M.; 8 Beatrice Lane, Wayne, NJ 07470 (US). LI, Bin; 41-09 75th Street, Elmhurst, NY 11373 (US). LIN, Xuren; 33-25 90th Street #2J, Jackson Heights, NY 11372 (US). ZHANG, Ju; 111 Mulberry Street #6H, Newark, NJ 07102 (US). (74) Agent: MARCOU, George, T.; Kilpatrick Stockton LLP, Suite 800, 700 13th Street, N.W., Washington, DC 20005 (US). | | (81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG). Published <i>With international search report.</i> <i>Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.</i> |

(54) Title: TRANSLATION OF MESSAGES TO AND FROM SECURE SWIFT FORMAT



(57) Abstract

The present invention facilitates handling of electronic messages between different computer networks and application programs by providing a message agent server system and a message format library system. The message agent server system (10) routes messages from application programs to external networks and external messaging systems. The message format library system includes message functions for formatting, translating, validating, reconciling, and/or parsing. In a financial services environment, the present invention facilitates communication of electronic messages by converting messages from a format used by a financial institution's network to the SWIFT (Society for Worldwide Interbank Financial Telecommunication) format.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

| | | | | | | | |
|----|--------------------------|----|---------------------|----|-----------------------|----|--------------------------|
| AL | Albania | ES | Spain | LS | Lesotho | SI | Slovenia |
| AM | Armenia | FI | Finland | LT | Lithuania | SK | Slovakia |
| AT | Austria | FR | France | LU | Luxembourg | SN | Senegal |
| AU | Australia | GA | Gabon | LV | Latvia | SZ | Swaziland |
| AZ | Azerbaijan | GB | United Kingdom | MC | Monaco | TD | Chad |
| BA | Bosnia and Herzegovina | GE | Georgia | MD | Republic of Moldova | TG | Togo |
| BB | Barbados | GH | Ghana | MG | Madagascar | TJ | Tajikistan |
| BE | Belgium | GN | Guinea | MK | The former Yugoslav | TM | Turkmenistan |
| BF | Burkina Faso | GR | Greece | | Republic of Macedonia | TR | Turkey |
| BG | Bulgaria | HU | Hungary | ML | Mali | TT | Trinidad and Tobago |
| BJ | Benin | IE | Ireland | MN | Mongolia | UA | Ukraine |
| BR | Brazil | IL | Israel | MR | Mauritania | UG | Uganda |
| BY | Belarus | IS | Iceland | MW | Malawi | US | United States of America |
| CA | Canada | IT | Italy | MX | Mexico | UZ | Uzbekistan |
| CF | Central African Republic | JP | Japan | NE | Niger | VN | Viet Nam |
| CG | Congo | KE | Kenya | NL | Netherlands | YU | Yugoslavia |
| CH | Switzerland | KG | Kyrgyzstan | NO | Norway | ZW | Zimbabwe |
| CI | Côte d'Ivoire | KP | Democratic People's | NZ | New Zealand | | |
| CM | Cameroon | | Republic of Korea | PL | Poland | | |
| CN | China | KR | Republic of Korea | PT | Portugal | | |
| CU | Cuba | KZ | Kazakstan | RO | Romania | | |
| CZ | Czech Republic | LC | Saint Lucia | RU | Russian Federation | | |
| DE | Germany | LI | Liechtenstein | SD | Sudan | | |
| DK | Denmark | LK | Sri Lanka | SE | Sweden | | |
| EE | Estonia | LR | Liberia | SG | Singapore | | |

TRANSLATION OF MESSAGES TO AND FROM SECURE SWIFT FORMAT

5 Field of the Invention:

 The present invention relates to systems and methods for facilitating electronic message transmission among different computer networks and application programs on the networks. The systems and methods are particularly advantageous for use in the transmission of electronic messages among financial institution networks and
10 application programs and the Society for Worldwide Interbank Financial Telecommunication (SWIFT) network .

Background:

 The use of electronic messaging in the financial services industry continues to
15 intensify. For example, due to evolution in the securities market, there is an ever increasing movement of funds into and out of international securities. This movement of funds is effected by electronic messages.

 Systems exist that permit securities traders to communicate electronically with each other. The Society for Worldwide Interbank Financial Telecommunications
20 (SWIFT) has developed standards for electronic messages routed over their financial network. Messages which meet the SWIFT standards are referred to herein as SWIFT messages. The use of SWIFT messages enables financial institutions to receive and process messages in a reliable and efficient manner.

 SWIFT messages are widely utilized in the international securities area as a
25 form of communication between brokers, clearing agents, financial institutions and other security transaction participants. One advantageous method for trading securities is described in US patent application serial number 08/700,836, filed August 21, 1996, entitled Method and Apparatus for Trading Securities
30 Electronically, the disclosure of which is hereby incorporated by reference. One embodiment of the methods described in this application includes the steps of: transmitting an order message from an originating broker workstation to a host computer in SWIFT format, the order message being directed to an executing broker

to buy or sell securities; and then transmitting a SWIFT format confirmation message from the executing broker to confirm the transaction.

Although SWIFT messages are extensively utilized in international securities transactions, the SWIFT message format is not generally utilized for message traffic on financial institutions internal servers. In addition, many application programs, utilized in the financial services area, are not capable of generating SWIFT messages. Therefore it would be advantageous to have methods for translating messages which are not in SWIFT format to SWIFT format. It would also be advantageous to provide application programs with the capability of generating SWIFT format messages through the use of a local set of functions.

The need to interface with international networks such as the SWIFT network, and to send and receive electronic messages, such as SWIFT messages, to such networks may present additional problems for financial institution networks in the routing and handling of message traffic. Thus, it would be advantageous to have a system for the routing of messages which includes the capability to receive messages, store messages, queue messages, validate the format of messages and route messages.

The foregoing advantages are achieved by the systems and methods of the present invention.

Summary of the Invention:

The present invention provides systems and methods that facilitate the handling of electronic messages (also referred to herein simply as "messages") by different computer networks and application programs. The present invention is particularly advantageous for use in worldwide financial institution computer networks and for use with application programs which access those networks. In a financial services environment the systems and methods of the present invention may be utilized to facilitate the communication of electronic messages, which are entered in a first messaging format utilized by a financial institution's network, to and from the Society for Worldwide Interbank Financial Telecommunication network which requires a defined format, referred to herein as "SWIFT", which may differ from the messaging format utilized by the financial institution.

In one aspect, the present invention provides a message exchange system. The message exchange system includes, as components, a message agent server; and a series of functions, collectively referred to herein as a "message format library", for applications which send and receive messages. The message agent server and message format library may be utilized together in an integrated system for the handling and transfer of electronic messages among discrete computer networks. The message agent server and the message format library may also be utilized independently.

The message agent server of the present invention may perform one or more of the following functions, routing messages, queuing messages; storing messages; receiving messages; logging messages; triggering applications which send and receive messages and validating the format of messages. Preferably the message agent server of the present invention will include all of the foregoing functions. In addition, the message agent server may include the message format library of the present invention. In this type of embodiment the message agent server may perform additional functions relating to the translation of messages.

An advantage of an embodiment of the message agent server of the present invention is that the message agent server may have the capabilities to receive, store, queue, validate and route messages.

The message agent server of the present invention may be connected to applications and other servers through TCP/IP connections. The routing functions of the message agent server may be utilized to messages created by an application to other applications on the same network, or through an interface out to different networks such as the SWIFT network. The message agent server may also be utilized to receive incoming messages from different networks and route the incoming messages to an application program.

The message format library feature of the present invention includes message agents for translating messages from a first format to another format such as the SWIFT format. The message format library may be advantageously utilized to take data elements from an application program and create electronic messages for sending to another party. The message format library may also be advantageously utilized to

parse incoming messages and recover data elements relevant to a particular application program.

The message agent server and message format library are utilized by application programs through local or remote function calls. In this regard, the message agent server is utilized by application programs to send and receive messages. The message format library may also be utilized by application programs, independently of the message agent server, to translate and manipulate messages received, and obtain data elements from the messages for processing within their own application's domain. As indicated above, the message format library may also be utilized by application programs to create electronic messages from data elements within the application program.

The services of the message agent server component of the present invention are performed without direct user involvement, although embodiments of the message agent server may include means for a network administrator to review message logs retained by the message agent server.

The services of the message format library component of the present invention may be accessed through a remote or local procedure call by an application program. The message format library may be accessed as part of a data entry system which includes a graphical user interface (GUI) for prompting input by the user of the application program.

The present invention also includes methods for the handling of messages utilizing the systems of the present invention.

The invention is described in the following sections with reference to particular types of messages which are utilized in conjunction with providing financial services. However, as will be understood by those of ordinary skill in the art, the concepts described herein apply to "message(s)" and/or "electronic message(s)" as used in a broad sense to comprise any type of message content; namely, the encapsulation of any data objects, including but not limited to: text, graphics, data, digitized voice or image, or the like; together with delivery, utilization and identification information that is needed to produce at the origin, and each final destination, those activities specified by the encapsulated content. In a preferred

embodiment described below, the present invention is utilized in message oriented data transmission.

As used herein the terminology "application program" or "program" is utilized to describe a computer program for accomplishing or effecting a particular function or functions, the computer program comprising a detailed and explicit set of directions (code) for accomplishing the function(s). For example, an application program could be written for accomplishing the global securities trading functions described above and in US patent application serial number 08/700,836 referred to above. Other examples of application programs utilized by financial institutions include programs for foreign exchange and money market trade confirmation matching for trades between corporations and their counterpart banks; programs for electronically distributing exchange rates; and portfolio management programs. In a broad context the systems and methods of the present invention are applicable to other application programs such as word processing programs, electronic mail programs, scheduling programs and the like.

In the detailed description of the invention reference is made to procedure calls and in particular to a remote procedure call (RPC) or a local procedure call (LPC). In many of the embodiments described below the system architecture is such that remote procedure calls (RPC's) are utilized by application programs and/or other system components. As will be understood by those of ordinary skill in the art, the present invention is not limited to the system architecture described in the embodiments below and in other forms of system architecture local procedure calls may be utilized.

Other terms are also utilized in manners consistent with their usage by those of ordinary skill in the art.

The systems and methods of the present invention are described in more detail below with reference to the following figures.

Brief Description of the Drawings:

Figure 1 is a block diagram illustrating a potential relationship among the message agent server, application programs, a gateway and the SWIFT network.

Figure 2 depicts the message flow of a message originated by an application program for transmittal to the SWIFT network. The embodiment of the system of the present invention depicted in Figure 2 includes a message format library.

5 Figure 3 depicts the message flow of an incoming message, for example a message originated on the SWIFT network.

Figure 4 depicts an overview of the functional components of an embodiment of a message agent server of the present invention which is linked via communication links to application programs and to an interface to an external network.

10 Figure 5 depicts the layered architecture of a possible embodiment of a message format service.

Figure 6 is a schematic of an embodiment of an application interface for use between a message agent server and application programs.

Figure 7 illustrates the architectural design of an embodiment of a message queue for use in a message agent server of the present invention.

15 Figure 8 depicts an embodiment of a message agent server architecture wherein the interface to a SWIFT network is provided by an X.25 interface.

Figure 9 depicts an embodiment of a state diagram for an outbound X.25 link server.

20 Figure 10 depicts an embodiment of a state diagram for an inbound X.25 link server.

Figure 11 depicts an alternative embodiment of a computing environment utilizing the system of the present invention.

Figure 12 depicts a data access model for a message agent server of the present invention.

25 Figure 13 is a representation of a graphical user interface for a message agent server system administration utility provided by an administration function of the system of the present invention.

Figure 14 is a representation of a graphical user interface for monitoring provided by an administration function of the system of the present invention.

30 Figure 15 is a representation of a graphical user interface for error display provided by an administration function of the system of the present invention.

Figure 16 is a representation of a graphical user interface for event details provided by an administration function of the system of the present invention.

Figure 17 is a representation of a graphical user interface for message services provided by an administration function of the system of the present invention.

5 Figure 18 is a representation of a graphical user interface for queued messages provided by an administration function of the system of the present invention.

Figure 19 is a representation of a graphical user interface for queued error messages provided by an administration function of the system of the present invention.

10 Figure 20 is a representation of a graphical user interface for an event search provided by an administration function of the system of the present invention.

Figure 21 is a representation of a graphical user interface for message statistics provided by an administration function of the system of the present invention.

15 Figure 22 is a representation of a graphical user interface for delayed/failed message statistics provided by an administration function of the system of the present invention.

Figure 23 is a representation of a graphical user interface for advanced message statistics provided by an administration function of the system of the present invention.

20 Figure 24 is a representation of a graphical user interface for FTS profiles provided by an administration function of the system of the present invention.

Figure 25 is a representation of a graphical user interface for inbound profiles provided by an administration function of the system of the present invention.

25 Figure 26 is a representation of a graphical user interface for inbound profile details provided by an administration function of the system of the present invention.

Figure 27 provides a graphic overview of an embodiment of a message format library of the present invention.

Figure 28 illustrates a tree built on a concatenation expression utilized in a message format library of the present invention.

30 Figure 29 illustrates a tree built on a parentheses pair expression utilized in a message format library of the present invention.

Figure 30 illustrates a tree built on an option pair expression utilized in a message format library of the present invention.

Figure 31 illustrates a tree built on a variable assignment expression utilized in a message format library of the present invention.

5

Detailed Description of the Invention:

As set forth above, the present invention includes a message agent server and a message format library which may be utilized in an integrated message exchange system for handling the exchange of messages among applications and computer networks.

10

According to the present invention a method for transferring electronic messages in a network, the network including a message agent server, a first terminal for communicating electronic messages in a first format, and a secure network, the secure network further connectable to at least a second terminal for communicating electronic messages in a secure network format, comprises: generating an electronic message at the first terminal in the first format; transmitting the electronic message from the first terminal to the message agent server; the message agent server automatically translating the electronic message from the first format to the secure network format; and transmitting the translated electronic message from the message agent server to the second terminal. The method may further comprise: generating a second electronic message at the second terminal in the secure network format; transmitting the electronic message from the second terminal via the secure network to the message agent server; the message agent server automatically translating the electronic message from the secure network format to the first format; and transmitting the translated second message from the message agent server to the first terminal.

15

20

25

The first terminal, second terminal and/or server may comprise a computer, for example a personal computer. The step of generating the electronic message at the first terminal in the first format may be performed using a software application such that the first format includes data formatted using the software application.

30

Examples of secure networks include financial networks wherein the secure network format, for example, is a SWIFT compatible format. The method may be

advantageously utilized wherein the first message includes a plurality of first message elements and the translated electronic message includes a plurality of secure network message elements, and the message agent server automatically parsing the message, such that the plurality of message elements correspond to the plurality of secure network message elements. In addition the method may include validating the electronic message or translated electronic message.

In embodiments of the method the second terminal may be disconnectable from the secure network and transmitting the translated electronic message may include connecting the second terminal to the secure network, the server automatically storing the translated message; the server automatically transmitting an alert message to the second terminal; and the second terminal retrieving the translated message.

In another aspect the present invention includes a method for transferring electronic messages between a first terminal, a message agent server and a first network comprising; generating an electronic message addressed to the first network on the first terminal; sending the message from the first terminal to the message agent server; interpreting the message so as to determine the address of the electronic message; and routing the message to the first network from the message agent server. The method may further comprise queuing the electronic message in the message agent server before it is routed to the first network. In addition, the step of routing may be delayed until the first network comes on-line such that the message will remain queued until the first network comes on-line.

The present invention also includes a system for transferring electronic messages in a network, the network including a message agent server, a first terminal for communicating electronic messages in a first format, and a secure network, the secure network further connectable to at least a second terminal for communicating electronic messages in a secure network format, the system comprising: means for generating an electronic message at the first terminal in the first format; means for transmitting the electronic message from the first terminal to the message agent server; means for automatically translating the electronic message from the first format to the secure network format; and means for transmitting the translated electronic message from the message agent server to the second terminal.

In another aspect, the present invention includes a system for transferring messages comprising: a message agent server comprised of: at least one queue; a processor which is used to at least determine the destination address of one of the messages and a plurality of communications links used to carry the messages; a first terminal; a first network coupling the first terminal to a first subplurality of communications links; and a second network coupled to a second subplurality of communications links. The first terminal, may be a server and may hold one of a plurality of application programs, for example a securities exchange program. The second network may be a world-wide network used to transfer financial messages. The queue may be used to store and transfer messages in a particular order destined for the first terminal and/or for the second network. Message transfer may be controlled for example by an MQ application.

In another aspect the present invention provides a message agent server used for transferring messages between a first terminal and a network the message agent server comprising: communications links coupling the message agent server to the first terminal and the network; a processor; a buffer for temporarily holding a message received via the communications links; wherein the processor operates under the control of an interpreting procedure so the processor determines the destination address of the message while the message is in the buffer; a first queue associated with the first terminal; a second queue associated with the network; where the message is transferred from the buffer to the appropriate queue based on the determination made by the processor under the control of the interpreting procedure.

In a further aspect the present invention provides a system comprising a message agent server coupled to a first network, the network including a first terminal, and coupled to a second network where the second network includes at least one terminal where the system is further comprised of: communications links coupling the message agent server to the first network and coupling the message agent server to the second network; where the message agent server is further comprised of a processor; message receiving means for receiving messages transmitted to the message agent server from the first and second networks via the communications links; means for interpreting the received messages which includes reading the address of the destination of the message; queuing means for storing the received

messages and for forwarding the messages to the communications links when told to do so; routing means for routing the messages to the appropriate destination based upon the determination of the means for interpreting. The means for interpreting the message may include means for validating the message format and/or means for
5 translating the message from a message format to a second message format, for example using a message format library. The queuing means may comprise a database and may include at least one of the message archive, retrieval, and re-transmission capabilities. The system of claim may further include means for monitoring message traffic and message agent server performance by an
10 administrator.

In a further aspect, the present invention provides a method for transferring electronic messages among a plurality of participants wherein the participants include a first terminal and a first network and the method uses a message agent server where the method comprises: preparing an electronic message on the first terminal addressed
15 to the first network; sending the message to the message agent server; interpreting the message to determine at least the message addressee; storing the message in a message queue assigned to the message addressee; triggering the first network to connect to the message agent server; and routing the message to the remote network after the remote network connects to the message agent server. The step of
20 interpreting the message may include validating the format of the message. In addition the method may further include monitoring message traffic.

In another aspect, according to the present invention, a message agent server for communication of messages among a plurality of participants in a computing environment, the participants including the message agent server; a remote application
25 residing on a first computer; and a remote network residing on a second computer, the participants being connectable by a physical network the message agent server comprising:

- a communications device for establishing a communications link;
- a processor, coupled to the communications device;
- 30 memory;
- message receiving means for receiving incoming messages;
- means for interpreting the messages;

queuing means for queuing and storing received messages; and

routing means for routing the message to the message addressee.

The means for interpreting the messages comprises means for determining the addressee of the message. In a preferred embodiment of the message agent server of the present invention the means for interpreting the messages includes means for
5 validating the message format, for example utilizing the message format library. The queuing means preferably comprises a database for queuing and storing messages. In a preferred embodiment the message agent server further comprises means for monitoring message traffic and message agent server performance by an
10 administrator. The monitoring means may also use the database utilized by the queuing means.

The message agent server may additionally include a failover of the message agent server components so that a failure in part of the message agent server does not prevent other portions of the message agent server from running. The failover
15 provision may preferably include a provision for redundant communication links to application programs or the network; redundant processors (CPUs) and redundant processes for receiving, queuing, routing, validating and other handling of messages. A message format utility which is used to maintain the various message formats and other meta data with a graphical user interface (GUI).

20 The message agent server may further include a message archive, retrieval, and re-transmission capability.

The message agent server may be constructed utilizing conventional computer hardware. Suitable processor hardware includes an Intel Pentium Pro processor or its equivalent. The computer operating system may be Microsoft NT or its equivalent.
25 As will be understood by those of ordinary skill in the art, the message agent server of the present invention may take advantage of increased processor speed and more robust operating systems as each becomes available in the future. The computing environment which includes the message agent server may comprise discrete computing platforms linked through TCP/IP connections. In this type of environment
30 the communication devices may comprise modems or the equivalent. The message agent server may also be utilized as an integral part of a local area or wide area network to handle message traffic among nodes on the network.

In a further aspect, according to the present invention, a method for transferring electronic messages among a plurality of participants in a computing environment, the participants including a message agent server; a remote application residing on a first computer; and a remote network residing on a second computer comprises:

5 preparing an electronic message in the remote application on a first computer, the message being addressed to the remote network residing on the second computer; sending the message to the message agent server; interpreting the message to determine at least the message addressee; 10 storing the message in a message queue assigned to the message addressee; triggering the remote network to connect to the message agent server; and routing the message to the remote network after the remote network connects to the message agent server. Preferably interpreting the message further comprises validating the format of the message, for example utilizing the message format library.

15 In a preferred embodiment the method of the present invention further comprises: monitoring message traffic on the message agent server.

The methods of the present invention may be performed on conventional computer hardware such as the hardware discussed above.

Further details regarding the message agent server and methods for 20 transferring electronic messages of the present invention are set forth below in the discussion of the appended Figures.

Figure 1 provides an overview of a computing environment which includes the message agent server of the present invention. As shown in Figure 1, a computing environment 2, may include a message agent server 10, having communication links 25 20, to application programs 12, 14 and 16. Communication links 20 may be direct network links, for example through an ethernet link, or dial-up links, using telecommunication links. The communication links are preferably TCP/IP links.

Application programs 12, 14 and 16 may be the same or different and may reside on a single computer or a plurality of computers. The application programs 30 will generally be message based, i.e. utilized for generating electronic messages for a particular purpose, although the present invention may be utilized with other types of application programs. An example of an application program utilized in the financial

services industry is the Global Clear program for the routing of orders and the confirmation of cross-border security trading, described above and in US patent application serial number 08/700,836. Other application programs utilized by the financial services industry include: programs for providing an electronic mechanism for matching confirmations for FX options trades between corporations/fund managers and their bank counter-parties; programs for matching confirmations of foreign exchange and money market trades between corporate customers and bank traders; programs for monitoring and controlling the valuation of collateral assets pledged by clients to secure transactions which include multiple traded and non-traded products; and programs for handling payment orders. Details of these application programs are set forth below.

As discussed above, the message agent server provides messaging and queuing capabilities for application programs. The message agent server is particularly advantageous for use with application programs that send and receive SWIFT messages. The application programs include programs utilized by the financial services industry such as the following: programs for providing an electronic mechanism for matching confirmations for FX options trades between corporations/fund managers and their bank counter-parties, referred to herein as FX Options Confirmation Matching; programs for matching confirmations of foreign exchange and money market trades between corporate customers and bank traders, referred to herein as FX Match; programs for monitoring and controlling the valuation of collateral assets pledged by clients to secure transactions which include multiple traded and non-traded products referred to herein as CCMP; programs for cross border routing and confirmation matching, referred to herein as Global Clear, and programs for handling payment orders. Further details relating to these application programs and the types of messages sent and received by each are set forth in the following paragraphs. Following the application program descriptions is a description of another embodiment of a system of the present invention described with reference to these particular application programs.

The FX Options Confirmation Matching application will provide an electronic mechanism for matching confirmations for FX options trades between corporations/fund managers and their bank counter-parties. In an embodiment of the

service, banks will transmit FX options confirmations (MT305 message format) to a message agent server over the SWIFT network, sometime after trading with a corporation using existing trading methods. The message agent server will safely queue the message, perform basic validation of the SWIFT format and header contents and return a positive or negative acknowledgment to the interface with the SWIFT network. Meanwhile, the FX Options system application program will have "subscribed to receive" messages addressed to it from message agent server when they are received from the SWIFT network. For each MT305 message it receives from message agent server, the FX Options system application program will invoke the message parsing service of message agent server, or the message format library, to break the message into its constituent fields, and store them in its database. The FX Options system application program will transmit the MT305 messages to the appropriate users, who will be provided with a graphical user interface (GUI) application which will be used to view and "affirm" the MT305 messages, as well print basic reports of the status of FX options confirmations and affirmation activities. Upon affirming the confirmations, the GUI application transmits them to the FX Options system application program, where they are stored in the database and sent to message agent server for delivery to the counter-party banks. Bank users will also be provided with a GUI application to view the status of FX options confirmations and print basic reports of the status of FX options confirmations and affirmation activities.

GlobalClear is a product for the routing of orders and the confirmation of cross-border security trading and is described in detail in US patent application serial number 08/700,836. referred to above. Messages of SWIFT MT500 series are sent to a GlobalClear server through GlobalClear application program client software direct link, file transfer and a SWIFT NETWORK INTERFACE. The messages are delivered through the same channels. The following SWIFT messages are handled: MT502; MT513; MT514; MT515; MT517; MT518; MT52x; and MT53x.

FX-Match matches confirmations of foreign exchange and money market trades between corporate customers and bank traders. It also enables on-line confirmation, monitoring of transaction status, storing historical information and provision of various reports.

Corporate customers enter information through a dial-up terminal or upload a file of FX and money market trade confirmations. Banks send information to FX-Match through the SWIFT NETWORK INTERFACE and file transfer. The file transfer may be performed using DECNET when originated internally from CITIBANK, and
5 KERMIT and other similar terminal protocols are used by external clients.

The message flow in FX-Match is simple and straightforward. Banks and corporate clients send MT300 and MT32x messages into FX-Match. FX Match matches them and sends confirmation messages. Corporate clients can also match the trades through direct online access to a terminal. Accepting a trade by a corporate
10 client causes a confirmation message to be sent to the counter-party bank via a SWIFT NETWORK INTERFACE. Banks may send trade cancellation confirmations by sending a cancel message through a SWIFT NETWORK INTERFACE. Corporate clients report canceled trades online, causing FX-Match to send a trade cancellation confirmation message to the counter-party bank through a SWIFT NETWORK
15 INTERFACE.

A cancel message for a MT300 message is a MT300 with field 22 containing the value "CANCEL". The same is true for cancellations of money market trades confirmations, e.g., MT32x. Banks currently send three types of messages: MT300; MT320 and MT324. Other types of messages may also be utilized including: MT301;
20 MT304; MT305.

CCMP is a platform which allows various bank businesses to monitor and control the valuation of collateral assets pledged by clients to secure transactions which include multiple traded and non-traded products. Product Processors send files daily containing all liability transactions to CCMP. CCMP nets the impact of the
25 transactions on each participant's collateral after being marked to market. CCMP then reports on these results to collateral monitoring units, customers, and risk and custody systems. In the case of a margin call, the customer can either initiate a message through a bank's internal network to SECORE (Global Custody System) or via SWIFT. A copy of the message is sent to CCMP when SECORE receives it. After
30 processing the collateral for acceptability, SECORE sends to CCMP a nightly position report of participants for reconciliation purposes.

CCMP handles the following SWIFT messages: MT520; MT521; MT522; MT523; MT530; MT531; MT532; MT533; MT592; MT100; MT900; and MT910.

Other application programs may utilize ISITC (Industry Standardization for Institutional Trade Communication) and FIX (Financial Information/Institution Exchange) message formats as well as the foregoing SWIFT message formats.

In the embodiment depicted in Figure 1 message agent server 10 is linked through communication links 20 to the SWIFT network 30. In other embodiments of the present invention, described in detail below, message agent server is linked through communication links to a gateway which in turn is linked to the SWIFT network and/or other networks.

In one possible use of the embodiment depicted in Figure 1 an electronic message is generated by an application program for transmission to the SWIFT network. The message is transferred via communication links to the message agent server. The message format allows the message agent server to identify the message as a SWIFT message and the message agent server stores the message in a SWIFT queue for transmission to the SWIFT network. After generation of a communication link to the SWIFT network the message is transferred to the SWIFT network. The generation of the communication link may be facilitated by the message agent server triggering the generation of the link.

Figure 2 depicts the flow of an outgoing message generated by an application program in an embodiment of the present invention. As shown in Figure 2, when an application program 12 sends an outgoing message for delivery to the SWIFT network it first prepares all the necessary variables and fields in a data entry step at a workstation 42. The data is converted in a remote or local message formatting procedure, 44 to form a SWIFT message. Preferably the message format library of the present invention is utilized in the formatting. The message is then passed to message agent server 10, through a communication link. Preferably the message agent server program is constantly listening for application programs on a communications link. Upon receipt of the message, the message format is validated, 46 and then the message is passed to a message queue 48. An interface program, 50 fetches the message from the queue and sends it through a communication link to the SWIFT network 30.

As noted above, the interface function may be incorporated within the message agent server. Alternatively, as shown in Figure 2, the interface may be separate from the message agent server and be reachable by the message agent server through a communications link. For example, the interface may form part of another computer network which is linked to the SWIFT network.

Figure 3 depicts the flow of an incoming message, for example a message from the SWIFT network according to an embodiment of the present invention. An incoming message from the SWIFT network, 30 is first received by interface 50, which may be incorporated within, or separate from, the message agent server. The message flows through the interface to the message agent server 10. The message is validated 46 and then passed to a queue 48 for delivery to an application. An application program 12 fetches the message from the queue via a communications link. The application issues a block reading for messages destined for it through a remote procedure call interface. The remote procedure either returns a message immediately or block-wait for the next arriving message. The message is then parsed and translated 44 from the SWIFT format for display in the application format. The translation may be accomplished by the message format library of the present invention. After translation, the message data is displayed at a workstation 42 for reading by a user. In alternative embodiments of the present invention the message translation may occur at the message agent server before the message is passed to the application program.

Messages may also be passed by the message agent server between and among different application programs.

In the embodiments shown in Figures 2 and 3, the system comprises two tiers. The first is the application program which takes user input and may reside on a client workstation. The second tier is the message agent server which controls and manages the data and business logic.

Embodiments of the message agent server of the present invention may perform a variety of services including one or more of the following services.

The message agent server may provide a link to the SWIFT network, permitting application programs to receive/send SWIFT formatted messages. The message agent server may provide transactional integrity of messages received by

application programs for transmission to a SWIFT NETWORK INTERFACE and for messages received from the SWIFT NETWORK INTERFACE bound for application programs.

5 An application program may use the message format library for formatting a collection of fields into SWIFT formatted messages.

The message agent server may validate SWIFT messages upon receipt from, and prior to transmission to, the SWIFT network.

10 The message agent server may provide SWIFT meta data for use by application programs to apply more meaningful names to SWIFT message data fields. This service may be used by an application program's graphical user interface for presenting the contents of a SWIFT message on a screen, using common language for the screen's field labels, e.g., "Transaction Reference Number" instead of the SWIFT field label, " :20:"

15 The message agent server may include failover of the message agent server components so that a failure in part of message agent server does not stop all of message agent server from running. This includes provision for redundant links to the SWIFT NETWORK INTERFACE, redundant processors (CPUs) and processes. The processes include the MQ-Network interface, MQ Series Engine and queues and the application-MQ interface RPC Server).

20 The message agent server may include a system administration utility which is used to administer and monitor the processes and message queues.

The message agent server and/or message format library may include a message format utility which is used to maintain the various message formats and other meta data with a graphical user interface (GUI).

25 The message agent server may include a message archive, retrieval, and re-transmission capability.

To support on-line/off-line use of the message agent server format libraries, the message agent server database on the client application program (PC) and the server may be synchronized at certain intervals.

30 The message agent server may support the SWIFT 97 message format.

The message agent server may include a set of functions as remote procedure calls that application programs, for example the GlobalClear application program,

may invoke to perform matching operations. These matching operations may include: matching one or more fields of messages of the same type, for example as with MT300; and/or matching one or more fields of messages of different types, for example as with GlobalClear's MT502 (order) and MT518 (execution).

5 The message agent server may include a set of functions as remote procedure calls for performing reconcilitaion operations.

Further message translation capabilities may also be included as part of the message agent server and/or message format library to translate a SWIFT message into another industry protocol message format.

10 Conceptually the message agent server may be utilized to handle file transmission and telex or fax transmissions.

The foregoing description and Figures 1, 2 and 3 are meant to provide a general overview of the role of the message agent server in a computing environment. The following paragraphs and Figures 4-12 provide a more detailed description of the features of the message agent server of the present invention.

15 Figure 4 depicts an overview of the functional components of an embodiment of a message agent server 10 of the present invention which is linked via communication links to application programs 12, 14 and 16 and to an interface 50 to an external network. As shown in Figure 4, a message agent server may include the following components.

20 A MQ (message queue)-Application Interface 100 which provides the interface between the persistent message queues 110 and the applications. In the embodiment depicted in Figure 4 the message agent server does not allow application programs 12, 14 and 16 to access the queues directly. The MQ-Application Interface serves as the broker between application programs and the message agent server message queues.

25 The MQ Application Interface allows application programs 12, 14 and 16 to send and receive messages using remote procedure calls. The application program interface between the application programs and the message agent server is described in detail with reference to Figure 5 and in general in the following paragraphs.

30 Suitable technologies for implementing the remote procedure calls include the Entera TCP RPC (remote procedure call) middleware which runs on a Windows NT

Server platform. The message agent server may also provide its services to older legacy applications running on VAX/VMS using either Entera DCE or TCP RPC, or a socket based message passing API and middle-ware.

5 The remote procedure call, for example the Entera RPC, may be used as the standard interface between the message agent server and application programs. The message agent server provides services as RPC calls in application programs such as, send-message and get-message that are implemented as RPC's by the message agent server. The MQ -Application Interface provides a non-polling mechanism to notify application programs of the receipt of messages after an application is on-line. At
10 application startup time, the application program may ask to receive all messages being held on its queue(s). Individual application programs will implement this capability in the manner best suited for their particular function.

Authentication may be utilized between the application programs and message agent server. When the application program signs up to use a queue everyday, an
15 authentication & security scheme may pass a security ticket to be used implicitly in subsequent RPC calls.

Application programs may be designed and implemented with many different remote procedure calls (RPC's) for accessing the message agent server. Preferably, the following remote procedure calls are provided.

20 Send Message: When an application program has a message to send out (e.g., to SWIFT) this RPC is called so that the MQ - Application Interface can safely queue the message for transmission to the target process.

25 Get Message: At startup and when it receives notifications from the MQ - Application Interface, the application program receives a message by calling this RPC.

30 Get Message Status: An application program may inquire as to the status of a message to determine for example, if an outbound message has been sent and received by its target process.

Re-Queue Message: The message agent server Monitor and Administration application, discussed below, will allow an operator to move messages from an error state to a queue to be transmitted, after an investigation and/or repair of the message.

5

Message Queues & Message Address Resolution: The message queues will comprise a queue for each application program: inbound messages. In addition there will be one generalized queue for inbound and outbound problem messages. Additional queues may be defined per client (such as for priority queues) and for general use.

10

The means by which message agent server determines which queues to place messages may be based on:

Application ID: Applications which use the message queues include the application programs and the interface application. Additional applications may be added and may be connected to the message agent server and use additional message queues. In addition, program applications may be able to send and receive messages with each other using the message agent server. Each application will have an Application ID, which is stored in a message agent server database. It is the responsibility of the application, not the MQ Application Interface, to provide information in the message to indicate its identity. A message agent server Translation Service API/RPC, using the message format library, will be provided to determine an application ID from the message and map it to a queue name.

15

20

25

Priority: Although the initial implementation of message agent server will process messages of equal priority, the message agent server may include a facility for an application to assign priorities to messages, e.g., high, medium, low and urgent.

30

Guaranteed Message Delivery: The message queuing system is configured such that messages handed to it by applications are not lost

by the message agent server. Once a message is handed to it by an application program, the message agent server may guarantee that the message is safely stored and that the message, once transmitted to its target, is guaranteed to have been transmitted and received. This means that messages should not be de-queued until acknowledgment of receipt has been made.

Asynchronous Message Delivery: Messages received by the message agent server for target application programs which are not on-line at the time of receipt are safely queued for delivery when the application program comes on-line.

Message Logging: All activities of queuing, de-queuing and re-queuing, receipt and transmission of messages are logged in the message agent server in a database. The log may contain the following information:

- DateTime of activity (YYYYMMDD-HH:MM:SS:CC)
- Activity
- Application ID of Sender
- Application ID of Target
- Message Contents.

Referring to Figure 4 the message agent server 10 further includes MQ (message queue) - Network Interface 120. The MQ - Network Interface is the component which provides the interface between the persistent message queues and interface 50 to an external network such as the SWIFT network. The MQ - Network Interface will be configured to support the network and communications protocol utilized by interface 50. The MQ - Network Interface also implements the queue processing of messages to be transmitted to SWIFT NETWORK INTERFACE, referred to as outbound messages, and those messages transmitted from SWIFT NETWORK INTERFACE to CrossMar client applications, referred to as inbound messages.

Message agent server 10 also includes Message Queuing System 130, for handling delivery of messages to and from the message queues 110. The message queuing system will interface with the MQ-Application Interface 100 and the MQ-Network Interface 120 and transfer messages back and forth from the message queues to these interfaces. The message queuing system may also interact with a message agent server monitor and administration system 140.

The message queuing system may utilize existing commercially available computer software to perform its functions. A preferred product for use as the message queuing system is the IBM MQSeries product. The MQ Series is a standard for message-oriented middleware, and the product is supported on a variety of platforms including VAX/VMS, Windows NT and HP-UX.

Also shown in Figure 4 is a message format library 140 which comprises a meta-data service 142; format service 144; parser service 146; and translation service 148. The message format library and the services it provides are described in detail in a following section, and covered generally in the next several paragraphs.

A basis for message handling as described above is the knowledge of message format. Knowledge of the message format is also relevant to message format validation. Therefore, the message agent server preferably includes access to message formatting services and information. The information on format may be extended to provide metadata service to application programs as a remote procedure call (RPC) or local procedure call through an application programming interface (API).

The meta-data, shown as 149 in Figure 4, may comprise a number of parts including the following:

- representation of SWIFT message formats;
- code tables for certain fields, e.g., currency and country code, valid BIC id, etc. and
- application specific interface captions on fields.

Using this information a graphical user interface (GUI) for application programs can be designed so that its appearance can be controlled dynamically from a centralized meta-data database.

For each message, the following information is maintained:
mandatory and optional fields;

ordering and repeating of fields; and
field format.

Application program specific requirements of formatted messages may also be supported in the meta-data database. This includes, but is not limited to the following aspects:

a generally optional field can be mandatory optional or absent (not present in interface) for a specific application;

a caption used on a GUI interface for each message, block and field;

a code table used for the field values;

client GUI display preference for each field such as radio box, jumbo box, or list box; and

read only protection.

Application programs 12, 14 and 16 may include features and characteristics which differ. To anticipate uneven formats and uneven format migration in the future, the message agent server provides format conversion, through message format service 144 so that the message agent server can continue the service even when data source, such as the application program and the data consumer, such as the SWIFT network, do not migrate formats at exactly the same time.

The message agent server parses messages, using parser service 146, into array of tagged fields and may be designed to pass messages to and from applications in the following format:

```
typedef struct {  
    small tag;  
    ptr, string] char *field_value;  
}  
Mas_field;  
typedef Mas_field Mas_message[].
```

As the industry trend is to have messages of various sources standardized to follow SWIFT messages, it is practical for message agent server to provide a message as a collection of fields labeled according to SWIFT labels. Preferably the message agent server supports the existing byte string message passing in the applications.

The message agent server may also include a message translation service 148. The translation service may be designed utilizing the procedures described below with reference to the message format library.

5 As set forth above, the message agent server may include a database to store message format information. Message format may be represented in generic relational database format. It may be implemented in SYBASE and or other relational databases. The format service design can be ported to any relational database easily. The layered architecture of a possible embodiment is illustrated in Figure 5.

10 The embodiment depicted in Figure 5 includes application program 12, which is a client on network with the network server referred to as application program server 15. Application program 12 accesses application program server 15 through a remote procedure call. Similarly, application program server 15 accesses the message agent server 30 through a remote procedure call.

15 Message agent server remote procedure calls (RPC's) may provide message parsing and formatting services to application programs linked to the message agent server. Application programs may also copy the message agent server format functionality to a local database 19, for example a Microsoft Access database. In this way, application programs may call local functions (through local procedure calls (LPC's) to parse and format messages when off-line. These local functions may be
20 implemented in a dynamic link library provided to the application programs linked to the message agent server.

The message agent server format service may access the format representation in the SYBASE database 35 through an RPC scheme set up by Entera or the representation in Microsoft Access database. The Microsoft Access database may be
25 synchronized to the SYBASE database manually.

Messages in each message type have the same header format, field structure and field format. A group of message types sharing the same header format and code tables form a message application version. The messages may be identified by their application version id, which is an integer, and their message id, which is an integer
30 too.

By way of example, the format representation may comprise as many as 17 tables as follows.

Table: MAS_MSG_VERSIONS

Table MAS_MSG_VERSIONS records the related dates of a message version and the overall format of messages in a version. SWIFT 1, SWIFT 11, SWIFT 96 are version examples.

5 Table: MAS_MSG_VARIABLES

This table lists all variables and names them.

Table: MAS_MESSAGES; MAS_GEN_BLOCK

Table MAS_MESSAGES contains a list of all messages for each version. MT 100, MT300, etc., are messages, Table MAS_GEN_BLOCK lists all the blocks.

10 Table: MAS_GEN_FORMAT; MAS_B_GEN_FORMAT

Table MAS_GEN_FORMAT and MAS_B_GEN_FORMAT contain a list of all fields for each message and other general attributes specified in message format standard documents.

The notion block is used to denote any set of consecutive fields. This is for the purpose of handling repeating and optional sequence. MAS_B_GEN_FORMAT is for fields in blocks.

Every application is assigned one or more application Ids (Aid's). All the following tables are for application specific format, hence contain the application ID as part of the reference keys.

20 Table: MAS_APPLICATION

This table contains all the applications and map application id to message version.

Table: MAS_MESSAGE_TITLE; MAS_BLOCK_TITLE

Table MAS_MESSAGE_TITLE contains a list of messages for each application ID. Table MAS_BLOCK_TITLE gives every block a title.

25 Table: MAS_APP_FIELD; MAS_B_APP_FIELD

Table MAS_APP_FIELD and MAS_B_APP_FIELD contain application specific information on fields. It also contains field format. We place field format in application specific data because different application might use different code tables while code table reference is specified in field formats

30

Table: MAS_CODE_TABLES

Instead of setting up multiple code tables, one table may be utilized to implement all code tables for the ease of implementation.

The following four tables implement cross-field constraints i messages.

Table: MAS_RULE_CLAUSE

5 This table implement propositions on individual fields. The following operators are used

E: Exists,
A: Absent,
M: Matches an expression,
10 X: Number within error,
0: Number bigger in a range,
C: Number smaller in a range.

Table: MAS_RULE_COMBINE

15 This table combines basic clauses with logical operators into composite clauses. A large set of logical operators may be utilized so as to save the number of intermediate composite clauses. The following operators are used:

A: And,
B: Both true or both false,
20 C: Left false and right true,
D: Both false,
M: Exclusive or,
N: Negation.

Table: MAS_RULE

25 Table MAS-RULE contains all the cross-field rules.

A table may also be utilized to represent the matching relationship useful in security trade confirmation.

Table: MAS_MATCH_TITLE

A list of individual sets of match rules.

30 Table: MAS_MATCH

Table MAS_MATCH contains the rules about which fields in one message should be matched to which fields in another message. Translation rules are represented in this table.

As explained in more detail below with reference to the message format
35 library, in an embodiment of the present invention the format expression for messages

is based on the regular expression of Norm Chomsky. A convention, similar to the convention utilized by the UNIX operating system and other operating systems may be utilized wherein:

- any literal character is prefixed with escape symbol '\';
- 5 'a' represents any character which can be used in a field;
- 'c' represents alphabetic characters;
- 'n' represents any digit or;
- 'd' represents any digit;
- 'r' represents new line;

10 Capital letters A, B, C, D, E, F, G and H are used for standard SWIFT formats on these format options;

<code_table_name> means any value from the table, <code_table_name length> includes the length of the string when the string length is fixed;

- 15 '(,)' are used to group items into one item;
- '[,]' group one or a number of items into an optional item;
- '|' means "or";
- prefixing a number repeats the item by up to as many times;
- prefixing a number then 'x' repeats the item on up to as many lines;
- 20 '**', instead of a number, represents arbitrary number;
- +',', instead of a number, represents arbitrary positive number;
- postfixing a number means exact number;
- postfixing 'x' then a number means exact as many lines; and
- number, '**' or '+' on optional sub-expression, 'r' <<>> or literal string is not
- 25 applied to express repetition.

Referring again to Figure 4, the message agent server 10 may also include a Message Agent Server (MAS) monitor and administration process 150 which includes a user interface. The administrative functions performed by the MAS Monitor and Administration Process include:

- 30 Starting and stopping the MQ - Application
- Setup and management of queues;
- monitoring of communication links;

examining the status of queues;
the querying and summarizing of messages according to criteria entered interactively;
management of user accounts and access rights;
5 examination of individual messages;
examination of service quality statistics such as maximum and average queue lengths, delays or exceptions counts;
operator intervention for exception handling; and
configuration tasks.

10 The MAS Monitor and Administration Process will preferably interact with a MAS Activity and Log Tables database 160. This database will include and store details relating to the services being performed by the message agent server. Further details regarding the administration process and user interface, and the database 160 are set forth below with reference to Figures .

15 Figure 6 is a schematic of an embodiment of an application interface for use between the message agent server and the application programs. As shown in Figure 6, an application program 12 may interface with a message agent server 10 through a remote procedure call (RPC).

The primary interface the message agent server provides for the application
20 programs may be based on RPC of DCE [] style. An RPC server framework with RPC stubs may be prepared with the stubs programmed in the framework to perform the functions advertised for the RPCs. These functions are covered above. The RPC server is constantly running to answer RPC calls. A thread is spawned for every RPC call. Working in the DCE/RPC paradigm, data elements are passed to and from MAS
25 as arrays of strings.

The application interface interacts with the message queing system 130 through client function calls and semaphores. The application interface interacts with a message agent server format library 120, through library function calls. The message agent server format library 120 may interact with a message format library
30 database 162 to retrieve informatin relating to message formats.

Preferably the message agent server handles all all RPC calls synchronously. Asynchronous processing in the application programs may be achieved by spawning threads in the applications.

The following functions are provided by the message agent server application interface:

5 MasParseData,
 MasFormatData,
 MasSendMsg,
 MasReceiveMsg,
 10 SentMsgStatus,
 RecentMsgStatus,
 MasGetErrors.

Whenever a message agent server RPC function succeeds, it returns a positive value. Otherwise, it returns a negative value which is the reference number for the errors
 15 encountered in the call. The application can call the MasGetErrors function to retrieve all the errors as another array of character strings.

The message agent server format functions MasParseData and MasFormatData are also provided as dynamic link library (DLL) functions together with the MasGetErrors function. The message agent server format functions may include
 20 necessary functions for passing SWIFT or other types of messages. Each application program identifies itself with one or more application identifiers (Aid) each of which corresponds to a unique header format and a set of messages in specific field format.

The message agent server format functions are described below. Similar functions are also described in detail below with reference to the message format
 25 library.

The MasParseData function is utilized to parse a message. A typical format for the function is as follows:

MasParseData Format:
 long MasParseData (
 30 [in] char Aid[],
 [in] char MessageText[],
 [out] char Msgld[],
 [out] char From.Bic[],
 [out] char ToBic[],
 35 [out] char GenTime[],

```

[out] char RecvTime[],
[out] char Priority[],
[out] char MUR[],
[out] char FieldNumber[],
5  [out] char FormatTag[],
[out] char FieldCaption[],
[out] char FieldContent[])
Remote and Local

```

10 This routine takes a formatted SWIFT message as input, and breaks it apart into the header fields, and other fields tags and values. It returns an error status if it is unable to parse the data correctly.

This routine is passed the Application Id in Aid, and a formatted swift message in MessageText. It pulls apart the message and breaks each of the header fields into
 15 MagId, FromBic, ToBic, GenTime, RecvTime, Priority,, and MUR. The fields are loaded into arrays FieldNumber, FormatTag, and FieldContent, with the number of fields placed into NumFields. An empty string in an output variable indicates that this optional variable is missing.

The Application Id is used to identify the format version. First the version
 20 format is loaded if it is not already in the memory. The function passes the header to get the message id and use it to locate, or load when not already in memory, the message body format. When a failure occurs, it will attempt to look for the next good field and resume parsing while registering an error.

The MasFormatData routine may be used as a local procedure call by an application
 25 program to validate and construct a SWIFT message. Message construction is done either by the client or the message is received over the SWIFT network. Formatting checks are preferably performed at the message agent server.. The function checks for mandatory fields, and verifies the values of all of the fields, including size. If an error is detected in parsing, the return value of the function is the Error Reference number
 30 that can be used to find the text errors corresponding to this message with a call to MasGetErrors.

A typical format for the function is as follows:

long MasFormatData

```

35  [in] char Aid[],
[in] char MsgId[],
[in] char FromBic[],
[in] char ToBic[],

```

```

[in]   char   priority[],
[in]   char   MUR[]
[in]   char   NumFields,
[in]   char   FieldNumber[][NumFields],
[in]   char   FormatTag[] [NumFields],
[in]   char   FieldContent[][NumFields],
[out]  char   MessageText[])
Remote and Local

```

10 The first parameter, Aid specifies the application Id that is requesting the function.
 The routine takes input parameters of all of the fields that comprise the SWIFT header
 (Parameters 2 to 6). The three arrays of string are the specific fields of the swift
 message. FieldNumber and FormatTag together form the field tag (i.e., 7 1 a). The
 corresponding array position in FieldContent is the value of that field. MessageText is
 15 the returned formatted Swift message including header, trailer and fields. NumFields
 contains the number of fields that passed into the routine. An empty string in input
 header variables indicates the optional variable is absent.

Although field order is significant for SWIFT, message agent server may be
 constructed to perform limited field sorting and thereby sort fields not in repeatable
 20 blocks, rather fields can be passed in random order except that the fields in repeatable
 blocks have to follow their order in their correspondent blocks.

The AppId and MsgId are used to identify the format version and body format
 which is loaded if it is not already in the memory. The variables are used to form the
 header and the fields to form the body. When a failure occurs, it will attempt to look
 25 for the next good field and resume formatting while registering an error.

The message agent server transmission functions are described in the
 following paragraphs.

The MasSendMsg routine will take a SWIFT message passed into it, and send
 the message out to the SWIFT network. It is passed in as a dynamically allocated string
 30 in Message. AppId is a constant pertaining to the application that is sending the
 messages.

A typical format for the function is as follows:

```

long MasSendMsg (
[in]   char   Message[],
[in]   char   AppId[],
[out]  char   MUR[])
Remote Only

```

Each SWIFT message may be given a unique (across all products) message identifier for transmission over the network. This identifier is returned in the variable MUR.

The format of MUR is "yyyymmddXXXnnnnn" where:

5 yyy is the year, and
 mm is the month, and
 dd is the date, and
 XXX is the application code (FXM, FXO, FXS, FXL, etc.), and
 nnnnn is the sequence number.

10 Another RPC function SentMsgStatus can be called with the parameter MUR to check the status of a sent message.

 The AppId can potentially trigger a translation before the message is queued. The MasSendMsg function calls MQPUT to place the message on a designated
 15 outgoing queue. It also puts the MUR and the status of a message in the status queue for all outgoing messages (QL_OUT_STATUS). It returns successfully after all the actions finish successfully.

 The MasReceiveMsg function is utilized by application programs to request a message addressed to the receiver identified by the AppId (Aid). It returns the first
 20 message in corresponding queue or the first to arrive in Message, i.e., the messaging mechanism is synchronous. The applications use other means to achieve asynchronous messaging.

 A typical format for the function is as follows:

25 long MasReceiveMsg (
 [in] char AppId[],
 [out] char Message[])
 Remote Only.

 The MasReceiveMsg function first attempts to decrement a semaphore guarding the corresponding receiving queue. After it passes the semaphore, it uses
 30 MQGET to retrieve the message and pass it back to the caller through the output variable. It will also put a new status of the message in the status queue for all incoming messages (QL_IN_STATUS).

 The message agent server may include utility functions such as the utility functions described below.

This SentMsgStatus routine takes the unique reference number, *MUR*, of a message previously sent by SendSwift and returns the status of the message in the output variable Status.

A typical format for the function is as follows:

5 Format: long SentMsgStatus (
 [in] char MUR[],
 [out] char Status[])
 Remote Only.

10 The SentMsgStatus routine will use the MUR as a key to retrieve the latest status of the sent message from the status queue for all outgoing messages (QL_OUT_STATUS).

15 The RecentMsgStatus function takes an application id (Aid) and returns all new message statuses since the last time this routine was called with the same application id. The function also returns the MUR of each corresponding message for each new status.

A typical format for the function is as follows:

 long RecentMsgStatus (
 [in] char Aid,
20 [out] char MUR[][],
 [out] char Status[][])

 Remote only

25 Every time the RecentMsgStatus routine is called, it gets the time stamp (format: yyyyymmdd) of the last call of the routine with the passed-in application id and the sequence number of the last retrieved message from an initiation file (mas.ini). Upon returning, it updates this information in the initiation file.

30 The MasGetErrors function is utilized to obtain ErrorReference numbers created during parsing or formatting of a message. If the parse or format functions return an error value, the number returned is the ErrorReference number. When this number is returned, the message agent server looks up the number and returns an array of strings that describes the errors that were encountered in the original call as text strings, with field number and format tag in the beginning of each line if necessary. Up to 32 errors may be returned at a time. The errors are returned in the variable ErrorDetail.

A typical format for this function is as follows:

```
long MasGetErrors
    [in] char ErrorReference[]
    [out] char ErrorDetail[][]
5 Remote and Local
```

Errors are placed in a circular buffer where MasGetErrors searches according to the ErrorReference number. After some time, depending on the buffer size and number of failed messages, the search is ended.

10 Figure 7 illustrates the architectural design of a message queue for use in a message agent server of the present invention. The message queue is described with reference to IBM's MQSeries message queueing engine, however the message agent server of the present invention may utilize other message queueing services to perform similar functions. The queues identified and described are examples of the types of queues which may be utilized. The system is further described with reference 15 to a SWIFT NETWORK INTERFACE which utilizes an X.25 type link. These description are provided by way of an example of an embodiment of a system of the present invention. As will be understood by those of ordinary skill in the art, further embodiments, and different functionality may be included in the message agent server, or the computing environment which includes the message agent server 20 without departing from the present invention.

The message agent server may provides messaging and queuing capabilities for financial service industry application programs such as the FX Options application program, the FX Match application program and the FXM ST (Foreign Exchange Statement Requests) application program. These applications receive and send SWIFT 25 messages through the message agent server which in turn communicates with a SWIFT NETWORK INTERFACE via an X.25 link. The message agent server may utilize IBM's MQSeries to facilitate sending messages to and receiving messages from the SWIFT NETWORK INTERFACE.

30 The overall design is depicted in Figure 7. The computing environment includes the FXM ST application program, 212, the FX Match application program 214 and the FX Option application program 216. The programs are linked through remote procedure calls (RPC's) to an MQ Series Server, 220. [Need description of

remaining components 231- of Figure 7. These components are described in greater detail in the following paragraphs.

The MQSeries allows Windows NT applications use message queuing to participate in message driven processing. Applications can communicate across different platforms by using the appropriate message queuing software products. For example, Windows NT and MVS/ESA applications can communicate through MQSeries for Windows NT and MQSeries for MVS/ESA respectively. The applications are shielded from the mechanics of the underlying communications.

MQSeries products implement a common application programming interface (message queue interface or MQI) whatever platform the applications are run on. This makes it easier to port applications from one platform to another.

With message queuing, the exchange of messages between the sending and receiving programs is asynchronous. This means that the sending and receiving applications are time- independent so that the sender can continue processing without having to wait for the receiver to acknowledge the receipt of the message. In fact, the target application does not even have to be running when the message is sent; it can retrieve the message after it is started.

On arriving on a queue, messages can automatically start an application using a mechanism known as triggering. If necessary, the applications can be stopped when the message or messages have been processed.

A queue manager manages the resources associated with it, in particular the queues that it owns. It provides queuing services to applications for Message Queuing Inter-face (MQI) calls and commands to create, modify, display, and delete MQSeries objects.

The message agent server queue manager (MASQM) may be created with the following attributes:

Default Queue Manager: YES
Dead letter queue: SYSTEM.DEAD.LETTER.QUEUE
Logging: CIRCULAR
Log file size: 1024K
Log path: C:\MQM\LOG.
(This log will preferably reside on a different drive from the queues.)
Primary logs: 5
Secondary logs: 2

The MQSeries command to create this Queue Manager is:

```
crtmqm /q /u SYSTEM.DEAD.LETTER.QUEUE /lc /lf 1024 /ld C:\mqm\log /lp 5 /ls
2 MASQM
```

- 5 This will create a queue manager that is the default queue manager for the particular machine, uses the queue SYSTEM.DEAD. LETTER. QUEUE for any undeliverable messages, uses circular logging, has 5 primary log files that are 4MB in size, and 2 secondary log files, all of which are located in C:\mqm\log.

- 10 The environment may be created with users defined specifically for MQSeries administration tasks. This user should have the allmqi permission assigned to their id or should be in the mqm group.

All of the processes that will be issuing MQI (message queue inquiry) calls should be started using user ids that are functional in nature so specific permissions on objects can be assigned accordingly.

- 15 Examples of queues and their purposes are listed below.

| | Queue Name | Purpose |
|-------------|-----------------|---|
| Requests. | QL_IN_FXOPT | Incoming Foreign Exchange Option Requests. |
| | QL_IN_FXMAT | Incoming Foreign Exchange Match |
| | QL_IN_FXSTM | Incoming Foreign Exchange Statement Requests. |
| INTERFACE . | QL_OUT_CTSW | Outgoing data back to SWIFT NETWORK |
| 25 | QL_IN_STATUS | Status queue for all incoming messages. |
| | QL_OUT_STATUS | Status queue for all outgoing messages. |
| | QL_OUT_STATUS_P | Pointers of status queue for all outgoing messages. |
| 30 | QL_ERROR_MAS | Error Queue used by the X25/MQSeries process and the MQSeries/RPC Server process. |
| | QI_INOUT_MAS | Initiation queue used to trigger an incoming or outgoing message event. |
| | QI_ERROR_MAS | Initiation queue used to trigger an error event |

- 35 The attributes of these queues are described in the following MQSC scripts to create them.

```
DEFINE QLOCAL('QL_IN_FXSTM') REPLACE +
DESCR('Foreign Exchange Statement Queue') +
PUT(ENABLED) +
DEFPRTY(O) +
5 DEFPSIST(YES) +
GET(ENABLED) +
MAXDEPTH(5000) +
MAXMSGL(25000) +
SHARE +
10 DEFLOPT(SHARED) +
MSGDLVSQ(FIFO) +
NOHARDENBO+
USAGE(NORMAL) +
TRIGGER +
15 TRIGTYPE(EVERY) +
TRIGDPH(1) +
TRIGMPRI(0) +
TRIGDATA("") +
PROCESS('QP_IN_FXSTM') +
20 INITQ('QI_INOUT_MAS') +
RETINTVL(999999999) +
BOTHRESH(0) +
BOQNAME("")+
SCOPE(QMGR) +
25 QDEPTHHI(80) +
QDEPTHLO(20) +
QDPMAXEV(ENABLED) +
QDPHIEV(DISABLED) +
QDPLOEV(DISABLED)
30 QSVCINT(6000) +
QSVCI EV(NONE)
```

```
DEFINE QLOCAL('QL_IN_FXOPT') REPLACE +
DESCR('Foreign Exchange Option Queue') +
35 PUT(ENABLED) +
DEFPRTY(O) +
DEFPSIST(YES) +
GET(ENABLED) +
MAXDEPTH(5000) +
40 MAXMSGL(6000) +
SHARE+
DEFLOPT(SHARED) +
MSGDLVSQ(FIFO) +
NOHARDENBO +
45 USAGE(NORMAL) +
TRIGGER +
TRIGTYPE(EVERY) +
TRIGDPH(1) +
```

- 40 -

```

    TRIGMPRI(0) +
    TRIGDATA("") +
    PROCESS('QP - IN - FXOPT') +
    INITQ('QI_INOUT - MASI) +
5    R-ETINTVL(999999999) +
    BOTHRESH(0) +
    BOQNAME("")+
    SCOPE(QMGR) +
    QDEPTHHI(80) +
10    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
    QSVCI(6000) +
15    QSVCI(0)

```



```

DEFINE QLOCAL('QL_IN_FXMAT') REPLACE
    DESCR('Foreign Exchange Match Queue') +
    PUT(ENABLED) +
20    DEFPRTY(O) +
    DEFPSIST(YES) +
    GET(ENABLED) +
    MAXDEPTH(5000) +
    MAXMSGL(6000) +
25    SHARE +
    DEFLOPT(SHARED) +
    MSGDLVSQ(FIFO) +
    NOHARDENBO+
    USAGE(NORMAL) +
30    TRIGGER +
    TRIGTYPE(EVERY) +
    TRIGDPHI(1) +
    TRIGMPRI(O) +
    TRIGDATA("") +
35    PROCESS('QP_IN_FXMAT') +
    INITQ('QI_INOUT_MASI) +
    RETINTVL(999999999) +
    BOTHRESH(O) +
    BOQNAME("")+
40    SCOPE(QMGR) +
    QDEPTHHI(80) +
    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
45    QDPLOEV(DISABLED) +
    QSVCI(6000) +
    QSVCI(0)

```

- 41 -

```
DEFINE QLOCAL('QL_OUT_CTSW') REPLACE +
    DESCR('Outgoing queue back to SWIFT NETWORK INTERFACE ') +
    PUT(ENABLED) +
    DEFPRTY(O) +
5    DEFPSIST(YES) +
    GET(ENABLED) +
    MAXDEPTH(5000) +
    MAXMSGL(6000) +
    SHARE +
10    DEFSOPT(SHARED) +
    MSGDLVSQ(FIFO) +
    NOHARDENBO+
    USAGE(NORMAL) +
    TRIGGER +
15    TRIGTYPE(EVERY) +
    TRIGDPTH(I) +
    TRIGMPRI(O) +
    TRIGDATA("") +
    PROCESS('QP_OUT - CTSW') +
20    INITQ('QI_OUT_MAS') +
    RETINTVL(999999999) +
    BOTHRESH(O) +
    BOQNAME("")+
    SCOPE(QMGR) +
25    QDEPTHHI(80) +
    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
30    QSVCIINT(6000) +
    QSVCIIEV(NONE)

DEFINE QLOCAL('QL_IN_STATUS') REPLACE +
    DESCR('Status queue for all incoming messages') +
35    PUT(ENABLED) +
    DEFPRTY(O) +
    DEFPSIST(YES) +
    GET(ENABLED) +
    MAXDEPTH(5000) +
40    MAXMSGL(32) +
    SHARE+
    DEFSOPT(SHARED) +
    MSGDLVSQ(FIFO) +
    NOHARDENBO+
45    USAGE(NORMAL) +
    TRIGGER +
    TRIGTYPE(DEPTH) +
    TRIGDPTH(5000) +
```

```

    TRIGMPRI(O) +
    TRIGDATA("") +
    PROCESS('QP_IN_STATUS') +
    INITQ('QI_INOUT_MAS') +
    RETINTVL(999999999) +
    BOTHRESH(O) +
    BOQNAME("") +
    SCOPE(QMGR) +
    QDEPTHHI(80) +
    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
    QSVCINT(6000) +
    QSVCI EV(NONE)

DEFINE QLOCAL('QL_OUT - STATUS-) REPLACE +
    DESCR(Status queue for all outgoing messages') +
    PUT(ENABLED) +
    DEFPRTY(O) +
    DEFPSIST(YES) +
    GET(ENABLED) +
    MAXDEPTH(5000) +
    MAXMSGSL(32) +
    SHARE +
    DEFSOPT(SHARED) +
    MSGDLVSQ(FIFO) +
    NOHARDENBO+
    USAGE(NORMAL) +
    TRIGGER +
    TRIGTYPE(DEPTH) +
    TRIGDP TH(5000) +
    TRIGMPRI(O) +
    TRIGDATA("") +
    PROCESS('QP_OUT_STATUS')
    INITQ('QI_INOUT_MAS') +
    RETINTVL(999999999) +
    BOTHRESH(O) +
    BOQNAME("") +
    SCOPE(QMGR) +
    QDEPTHHI(80) +
    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
    QSVCINT(6000) +
    QSVCI EV(NONE)

```


- 43 -

```

DEFINE QLOCAL('QL_ERROR - MAS') REPLACE +
    DESCR('Local Error Queue') +
    PUT(ENABLED) +
    DEFPRTY(O) +
5    DEFPSIST(YES) +
    GET(ENABLED) +
    MAXDEPTH(5000) +
    MAXMSGL(250) +
    SHARE +
10    DEFSOPT(SHARED) +
    MSGDLVSQ(FIFO) +
    NOHARDENBO+
    USAGE(NORMAL) +
    TRIGGER +
15    TRIGTYPE(EVERY) +
    TRIGDPH(I) +
    TRIGMPRI(O) +
    TRIGDATA("") +
    PROCESS('QP_ERROR_MAS') +
20    INITQ('QI_ERROR_MAS') +
    RETINTVL(999999999) +
    BOTHRESH(O) +
    BOQNAME("") +
    SCOPE(QMGR) +
25    QDEPTHHI(80) +
    QDEPTHLO(20) +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
30    QSVCINT(6000) +
    QSVCI EV(NONE)

DEFINE QLOCAL('QI_INOUT_MAS') REPLACE +
    DESCR('Initiation queue for incoming and outgoing message events') +
35    PUT(ENABLED)      +
    DEFPRTY(O)         +
    DEFPSIST(YES)      +
    GET(ENABLED)       +
    MAXDEPTH(5000)     +
40    MAXMSGL(6000)    +
    SHARE              +
    DEFSOPT(SHARED)    +
    MSGDLVSQ(FIFO)     +
    NOHARDENBO         +
45    USAGE(NORMAL)    +
    NOTRIGGER          +
    TRIGTYPE(FIRST)    +
    TRIGDPH(1)         +

```

- 44 -

```

    TRIGMPRI(O)      +
    TRIGDATA("")     +
    PROCESS("")      +
    INITQ("")        +
5   RETINTVL(999999999) +
    BOTHRESH(0)      +
    BOQNAME("")      +
    SCOPE(QMGR)      +
    QDEPTHHI(80)     +
10  QDEPTHLO(20)     +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
    QDPLOEV(DISABLED) +
    QSVCCINT(999999999) +
15  QSVCCIEV(NONE)

    DEFINE QLOCAL('QI_ERROR_MAS') REPLACE +
    DESCR('Initiation queue for ERROR events') +
    PUT(ENABLED)      +
20  DEFPRTY(O)        +
    DEFPSIST(YES)     +
    GET(ENABLED)      +
    MAXDEPTH(5000)    +
    MAXMSGSL(6000)    +
25  SHARE             +
    DEFSOPT(SHARED)   +
    MSGDLVSQ(FIFO)    +
    NOHARDENBO        +
    USAGE(NORMAL)     +
30  NOTRIGGER         +
    TRIGTYPE(FIRST)   +
    TRIGDPTH(1)       +
    TRIGMPRI(O)       +
    TRIGDATA("")      +
35  PROCESS("")       +
    INITQ("")         +
    RETINTVL(999999999) +
    BOTHRESH(0)       +
    BOQNAME("")       +
40  SCOPE(QMGR)       +
    QDEPTHHI(80)      +
    QDEPTHLO(20)      +
    QDPMAXEV(ENABLED) +
    QDPHIEV(DISABLED) +
45  QDPLOEV(DISABLED) +
    QSVCCINT(999999999) +
    QSVCCIEV(NONE)

```

The foregoing scripts are provided by way of example. Alternative scripts may be utilized to achieve similar results.

The processes and their attributes are listed below.

| | | |
|----|--------------|---|
| 5 | Process name | Function |
| | QP_IN_FXSTM | Triggered when an FXMatch Statement message arrives. |
| | QP_IN_FXOPT | Triggered when a FX Option message arrives. |
| | QP_IN_FXMAT | Triggered when a FX Match message arrives. |
| 10 | QP_OOUT_CTSW | Triggered when a messages is to be sent back to the SWIFT Network Interface |
| | QP_ERROR_MAS | Triggered when an error event occurs. |

The MQSC script to create these queues is as follows.

```

15
DEFINE PROCESS('QP - IN_FXSTM') REPLACE +
    DESCR('This process will be triggered for FXSTM activity') +
    APPLTYPE(WINDOWSNT) +
    APPLICID('c:\masq\programs\incsem\incsem.exe') +
20    USERDATA('DEBUG=1') +
    ENVRDATA("")

DEFINE PROCESS('QP_IN_FXOPT') REPLACE +
    DESCR('This process will be triggered for FXOPT activity') +
25    APPLTYPE(WINDOWSNT) +
    APPLICID('c:\masq\programs\incsem\incsem.exe') +
    USERDATA('DEBUG=1') +
    ENVRDATA("")

30    DEFINE PROCESS('QP_IN_FXMAT') REPLACE +
        DESCR('This process will be triggered for FARHAT activity') +
        APPLTYPE(WINDOWSNT) +
        APPLICID('c:\masq\programs\incsem\incsem.exe') +
        USERDATA('DEBUG=1') ENVRDATA("")
35

DEFINE PROCESS('QP_ERROR_MAS') REPLACE +
    DESCR('This process will be triggered for ERROR activity') +
    APPLTYPE(WINDOWSNT) +
    APPLICID('c:\masq\programs\geterror\geterror.exe') +
40    USERDATA('DEBUG=1') +
    ENVRDATA("")

DEFINE PROCESS('QP_OUT_CTSW') REPLACE +
    DESCR('This process will be triggered for outgoing activity') +
45    APPLTYPE(WINDOWSNT) +

```

- 46 -

```

    APPLICID('c:\masq\programs\incsem\incsem.exe') +
    USERDATA('DEBUG=1') +
    ENVRDATA("")

```

5 The foregoing scripts are provided by way of example. Alternative scripts may be utilized to achieve similar results.

MQI channels connect an MQI client to a queue manager on a server machine. It is for the transfer of MQI calls and responses only and it is bi-directional. A channel definition

10 exists for each end of the link.

The channels and their attributes are listed below.

| | | |
|----|--------------|--|
| | Channel Name | Purpose |
| | QC X25SVR | MASQM Server Connection to X25 Server (an MQSeries Client) |
| 15 | QC_X25SVR | MASQM Client Connection from X25 Server (an MQSeries Client) |

The MQSC script to create these queues is as follows.

```

20       DEFINE CHANNEL('QC_X25SVR') +
         CHLTYPE(SVRCONN) +
         TRPTYPE(TCP) +
         MCAUSER("") +
         DESCR('MASQM Server Connection to X25 Server (an MQ Client)')
25
         DEFINE CHANNEL(QC - X25SVR)+
         CHLTYPE(CLNTCONN) +
         TRPTYPE(TCP)      +
         CONN AME(I 92.193.83.149) +
30       QMNAME(MASQM)      +
         DESCR('MASQM Client Connection from X25 Server (an MQ Client)')

```

As described above, a triggering function may be implemented to trigger a call to a queue when a message is waiting for delivery or transmission. The triggering processes may be implemented in a batch file utilizing the following protocol.

35 Step 1: Start the listener for MASQM to use TCP/IP and to listen on Port 1414 (default port).

Command: start runmqslr /t tcp /p 1414 /in MASQM

The preceding "start" causes the MS-DOS shell to spawn another shell to run the actually command so that commands may continue to be issued in the same MS-DOS shell.

Step 2: Start the trigger monitor to wait in the initiation queue for a trigger message.

5 Command: runmqtrm. /m MASQM /q QI_INOUT_MAS

The output from all of these processes may be sent by using normal output redirection methods.

After these processes are running, when a message is sent to QL_IN_FXMAT, QL_IN FXOPT, QL_IN_FXSTM, or OL OUT CTSW, the appropriate trigger will fire and
10 the program incsem.exe will be invoked. The program will parse the trigger message and increase the appropriate semaphore so the message agent server server will be signaled. The message agent server will then connect to the appropriate queue manager, open the appropriate queue, and read all of the messages until the queue is empty. At this point the program will return and wait for the next trigger to fire.

15 The protocol for troubleshooting the triggering function may include the following items. Checking the local queue definition to ensure that the NOTRIGGER parameter is not set, and that the PROCESS and INITQ fields have the correct values. Checking to ensure that the INITQ and PROCESS objects exists. The commands are:

runmqsc <QueueManager>
20 dis q(initQueueName)
dis pro(processName) all

The output from the display process command may be viewed to ensure that the path to the triggered program is correct, and that the program is there. The state of the trigger monitor may also be reviewed to ensure that the trigger monitor in running,
25 and connected to the correct initiation queue for the appropriate Queue Manager.

If the notion of DEBUG has beent coded and passed to the triggered program, the process may be alter to turn DEBUG on and send a message and the result reviewed.

The command to alter the process is:

30 runmqsc
alter process(processName) userdata('DEBUG=1').

The trigger system may also be troubleshot may using the trigger type of FIRST and making sure the queue is empty before A user start the trigger monitor (see runmqtrm command). A trigger type of DEPTH may also be utilized. When the DEPTH triggers fires, the queue manager will set the NOTRIGGER attribute. It is the responsibility of the triggered application to issue the MQSET call to reset the TRIGGER attribute.

The message agent server may include security features. Security in MQSeries for Windows NT uses the users and groups that are created in the "User Manager Facility" from the Windows NT "Administrative Tools" icon. If a user Id belongs to the "mqm" (IBM MQSeries Administration Group) A user will have all of the authorities to all resources. There are situations, such as production environments that the more secure features of MQSeries will need to be in place, to ensure protection of system resources as well as protection of data.

The command used to establish specific permissions against MQSeries objects is setmqaut. A user must be in the mqm group to use this command. The parameters that follow this command are listed below.

setmqaut /in Queue Manager Name (Queue Manager Name that owns the object)

/n Object Name (Name of the Queue or Process. Not needed if the Object Type is qmgr)

/t Object Type (queue or q, process or prcs, qmgr, must be lower case)

/p Principal Name (Name of the principal (ID) to which the authorizations are to be granted. A user can enter more than one, but each must be prefixed by the /p flag)

/g Group Name (Name of the group (ID) to which the authorizations are to be granted. A user can enter more than one, but each must be prefixed by the /g flag) Authorizations are listed in the table below. To add an authority, prefix it with a '+', to remove one prefix it with a '-'.

| Authority | Queue | Process Manager | Queue |
|-----------|-------|--------------------|-------|
| all | x | X | x |
| alladm | x | X | X |

- 49 -

| | | | | |
|----|---------|---|---|---|
| | allmqi | X | X | x |
| | altusr | | X | |
| | browse | x | | |
| 5 | chg | x | x | x |
| | clr | x | | |
| | connect | | | x |
| | crt | X | X | x |
| | dlt | x | x | x |
| | dsp | X | x | X |
| 10 | put | X | | |
| | inq | x | x | x |
| | get | x | | |
| | passall | x | | |
| | passid | x | | |
| 15 | set | x | X | x |
| | setall | x | x | |
| | setid | x | x | x |

Example: setmqaut /in queue.manager. 1 /n QL_IN_FXMAT /t queue /p binli +put

20 These permissions will allow the userId binli the ability to issue the MQPUT or MQPUT I call on the object QL_IN_FXMAT.

Once authorities have been assign to the appropriate groups, they can be displayed using the dspmqaut command. The parameters that follow this command are listed below.

25 dspmqaut /m Queue Manager Name (Queue Manager Name that owns the object)

/n Object Name (Name of the Queue or Process. Not needed if the Object Type is qmgr)

/t Object Type (queue or q, process or prcs, qmgr, must be lower case)

30 /p Principal Name (Name of the principal (ID) to which the authorizations are to be granted. A user can enter more than one, but each must be prefixed by the /p flag)

/g Group Name (Name of the group (ID) to which the authorizations are to be granted. A user can enter more than one, but each must be prefixed by the /g flag)

35 Example: dspmqaut /in MASQM /n QL_OUT_CTSW /t queue /p binli

Output from the display command above is shown below.

Entity binli has the following authorizations for object QL_OUT_CTSW:

get

- 50 -

browse

put

inq set crt dlt chg dsp passid passall setid setall clr

5 Queue Manager permissions may be implemented and are preferably controlled with careful consideration. All processes that will be issuing MQI interface calls, are preferably started using a functional ID or GROUP for that process. This ID or GROUP will need the ability to issue an MQCONN call, to connect to the Queue Manager. The command to enable this connection is shown below:

setmqaut /in queue.manager. I /t qmgr /p binli +connect

10 This command would allow a process started with the binli userId access to connect to the Queue Manager.

The other permissions on the Queue Manager should be assigned to an entirely different group, that will be used for administrative purposes. Especially chg (Change), crt (Create), dlt (Delete) and any of the all* authorities.

15 Queues are the facility used to store data. Whether persistent or non-persistent, the data is preferably protected from intrusion, or accidental access. Each application that is running on a given platform, can be limited to the objects it can use and what functions it can perform on these objects.

20 A process is an event specified to occur when the trigger criteria for a local queue has been satisfied. The commands for setting and revoking these permissions follow the same conventions as Queue Managers and Queues.

The message agent server MQ/Series server may be installed and deployed on a computer platform utilizing the following steps:

Step 1. Action: install MQSeries in C:\mqm

25 Explanation: C:\mqm is the recommended directory

Step 2. Action: copy C:\masq from a development machine

Explanation: including all subdirectories

Step 3. Action: run C:\masq\commands\genBatch.exe

Explanation: genBatch creates all the NT batch files used in the steps below.

30 While running, A user is asked to specify root directory for the following:

MQ Series root directory [C:\mqm]: X:\mqm

MAS Queue Manager root directory [C:\masq]: X:\masq X is the drive letter that installer specified. All batch file are generated *in X:\masq\commands\.

Step 4. Action: run C:\masq\commands\crtmasqm. bat

Command: crtmqrn /q /u SYSTEM.DEAD.LETTER. QUEUE /lc /lf 1024

5 Ad

C:\mqm\log /lp 5 /ls 2 MASQM

Explanation: queue manager MASQM is created

Step 5. Action: run C:\masq\commands\stmasqm. bat

Command: strmqm

10

Explanation: queue manager MASQM is started

Step 6, Action: run C:\masq\commands\crtdefob.bat

Command: runmqsc < C:\mqm\mqsc\amqscoma.tst >

C:\mqm\mqsc\defobj.out

Explanation: default and system objects (default channel definitions, etc.) are
15 created

Step 7. Action: run C:\masq\commands\crtmasob.bat

Command: runmqsc < C:\masq\objects\masqm\masqmobj.in > C:\masq\obj
ects\masqm\masqmobj. out

Explanation: creates all the MASQM objects (queues, processes, channels)

20

Step 8. Action: copy C:\mqm\qmgrs\masqm\@ipcc\amqclchl. tab to C:\mqm

Explanation: the client channel definition file needs to be in DefaultPrefix
directory defined in C:\mqm\mqc.ini, otherwise, environment variables MQCHLLIB
and MQCHLTAB have to be set (another alternative is to use the MQSERVER
environment variable which overrides the other two)

25

Step 9. Action: run C:\masq\commands\stmasqml.bat

Command: start runmqlsr /t tcp /p 1414 /m MASQM

Explanation: starts a listener on MASQM

Step 10. Action: create shortcuts to stmasqm.bat and stmasqml.bat and put
them in the startup folder

30

Explanation: so they will be automatically started on NT restart

Preferably a message agent server of the present invention will include
methods for handling errors, including message delivery errors, message format errors

and the like. A possible means for error handling is set forth in the following paragraphs.

All message agent server errors may be put in a local queue QL_ERROR_MAS. A trigger may then be fired every time a message is put on this queue. A trigger program GetError.exe will in turn be invoked to get the error message and post an entry in the NT event log (viewable from NT event viewer). Preferably the GetError process is triggered for every message that arrives on the QL_ERROR_MAS queue.

By way of example, the message agent server may be configured with three categories of errors, namely MAS_INFO, MAS_WARN, and MAS_ERROR. These categories are defined in the following script used by NT event log.

```
MessageIdTypedef=WORD
MessageId=0x1
SymbolicName=MAS_INFO
Language=English
Category #1
```

```
MessageId=0x2
SymbolicName=MAS_WARN
Language=English
Category #2
```

```
MessageId=0x3
SymbolicName=MAS_ERROR
Language=English
Category #3
```

```
MessageIdTypedef=DWORD
MessageId=0x100
Severity=Informational
Facility=Application
SymbolicName=INFO_ONE
Language=English
This is an informational message.
```

```
MessageId=0x200
Severity=Warning
Facility=Application
SymbolicName=WARN-ON-E
Language=English
This is a warning message.
```

```
MessageId=0x3000
```

Severity=Error
Facility=Application
SymbolicName=ERROR-ONE
Language=English
5 This is an error message.

These errors are provided by way of example. The message agent server may include functionality for handling different types of errors depending on the needs of the application programs and network interface.

10 As described above, the message agent server includes a network interface to allow messages to be passed to an external network such as the SWIFT network. The interface or gateway to the SWIFT network may be included within the functionality of the message agent server as a local function, ie residing on the same server (computer hardware). Alternatively, the the message agent server may interact with
15 the interface or gateway through a communications link.

Figure 8 depicts an embodiment of a message agent server architecture wherein the interface to a SWIFT network is provided by an X.25 interface. As shown in Figure 8, a computing environment may include the FX Options application program 216 and the GlobalClear application program 218 which each link to
20 message agent server 10, through a remote procedure call to the application interface 100. The application interface 100 is linked to message queue manager 130, which manages the message queues 110. A message queue network interface 120 is provided to interface between the message agent server and the X.25 network 45 which includes a SWIFT NETWORK INTERFACE 50. The message agent server is
25 also linked to a message format library 145. The application programs, application interface, message queue manager, message queues and message format library are described in preceding sections. The following paragraphs describe a possible embodiment of a SWIFT NETWORK INTERFACE utilizing an X.25 network connection.

30 The message queue network interface Connection Module is one of the integrated components of Message Agent Server (MAS). This module may utilizes EiconCard technology to provide X.25 interface thus enabling applications to exchange messages with an X.25 network through message agent server. The

interface may also handle message security, exception handling/recovery, and much more.

The X.25 Interface Between message agent server and SWIFT NETWORK INTERFACE may be implemented as follows. The application level protocol
5 between the message agent server and the SWIFT NETWORK INTERFACE may follow the CMX's X.25 interface standard, which provides a message transfer mechanism between CMX system and a remote DTE like message agent server through X.25 network. Alternatively the 'interface between message agent server and CMX operates on Switched Virtual Circuit (SVC).

10 By way of overview, in a CMX protocol, a circuit is established to enable message transfer in single direction, either SEND ONLY or RECEIVE ONLY. Based on this assumption, two virtual circuits must be established to provide basic message transfer capability between message agent server and the SWIFT NETWORK INTERFACE.

15 One circuit is to allow the message agent server to send messages to the SWIFT NETWORK INTERFACE. The SWIFT NETWORK INTERFACE will send "ack" or "nak" back to the message agent server over the same channel. This channel is the SEND ONLY channel for message agent server.

20 Another circuit is to allow the message agent server to receive message from the SWIFT NETWORK INTERFACE. The message agent server will send ack or nak back to the SWIFT NETWORK INTERFACE over the same channel. This channel is the RECEIVE ONLY channel for message agent server.

25 These two channels may use the same DTE address pair between the message agent server and the SWIFT NETWORK INTERFACE. During normal operations, the two channels will generally not interfere with each other.

The CMX protocol also states that the SENDER is responsible to establish the network connection and to retry the connection if the link is disconnected.

When the message queue network interface starts, it may take the following initialization steps:

30 Initialize event queue for logging system events.

Make sure EiconCard has started and is running on the machine this module runs on.

Test which applications are running.

Initialize application classes and various structure array.

If any of above steps fail, this module will report exception and stop itself.

5 After the initialization steps pass smoothly, the X.25 server module spawns two link threads, 121 and 123: one manages the inbound channel, the other takes care of outbound channel.. The server module terminates after both link threads are terminated.

Each thread takes its own initialization steps:

Check which applications it need to deal with.

10 Initialize an array of application handles.

Load the table about mapping among applications, queues, and semaphores.

This mapping table must be persistent.

Make connection to message agent server Message Queuing System (MQ Manager) and open the error queue. It requires that MQ Manager be running already.

15 For each application the thread deals with, it opens semaphores. If the semaphores for the application are not there, it will create them. Note that the message agent server MQ-application interface RPC Server also create the semaphores, so if the RPC Server is running, the link thread just needs to open the semaphores.

Make connection to SWIFT NETWORK INTERFACE .

20 If the MQ Manager or message agent server RPC Server is not started, the above initialization steps will fall, and the link thread will terminate itself If everything works fine, the link thread will begin its normal operation.

25 There are a certain number of protocol stages that the link thread runs into. At each stage the link thread should call certain application 'interface functions based on what application the handling message belongs to.

When link thread wants to shutdown itself, it does the following:

Close the VC connection to SWIFT NETWORK INTERFACE ,

For each application the thread interacts with, close semaphores and all queues, but don't touch the messages.

30 Close the error queue, and disconnect from message agent server queue manager.

Unload the mapping table among applications, queues, and semaphores. Make persistent change if changes are needed.

During initializations, if anything fails the X.25 server or its threads just terminate themselves.

5 As set forth above, the Message Queuing System may utilize a set of queues configured by MQSeries.

For each message flow direction, Queue Manager provides a status queue to report message transfer status. The status messages in these queues has the following format:

10 yyyymmddXXXNNNNnnnnnnS

| <u>Format code</u> | <u>Explanation</u> | <u>Example</u> |
|--------------------|--------------------------|----------------|
| yyymmdd | date of message transfer | 19970110 |
| XXX | message incoming source | FXO, SWT |
| NNNN | SWIFT session number | 0392 |
| nnrLn | SWIFT sequence number | 174038 |
| S | status code | P, S, F |

15

Possible values for the status code are: P(ending), S(uccess), and F(ail).

Implementation of the message functionality may be accomplished as follows. For an outbound message:

20 The Message Queuing System puts outgoing message onto queue QL_OUT_CTSW.

When the Message Queue Network Interface module gets the message from QL_OUT_CTSW, it will transfer it out to the X.25 network. Based on the result of transfer, it will put a status message back to the outbound status queue

25 QL_OUT_STATUS with the message reference number and the status code. The status code is either S or F. The Message Queuing System gets the status message and process it.

For an inbound message: When the Message Queue Network Interface module get a message from X.25 network, it will put it onto appropriate queue based on the application it belongs to. It will also construct a status message and put it onto the
30 inbound status queue QL_IN_STATUS with the message reference number and the status code. The status code is P for pending status. The Message Queuing System gets

the application message. After the message is transferred to application, the Message Queuing System will update the status of the status message.

The foregoing functionality may be accomplished through the use of modules including an Outbound Service Module and an Inbound Service Module. These
5 modules may be part of the message queue network interface.

The Outbound Service Module is responsible for sending application messages out to X.25 network over the SEND CHANNEL. Basically this module is in one of the following three states:

1. CALL State

10 When the Outbound Service Module starts, it's in this state by default. The SENDER will make a call request to connect to the remote SWIFT NETWORK INTERFACE DTE. The connection is expected to stay up until the system shutdown. If the call request fails, the sender will retry the connection at regular interval until the call is successfully accepted by the remote RECEIVER.

15 2. SEND_MSG State

Once the call is established, the sender can start sending messages. For every message directed to SWIFT NETWORK INTERFACE, it may follow this procedure:

Get message from the Output Queue (QL_OUT_CRSW) but do not remove it from the queue.

20 Send message to SWIFT NETWORK INTERFACE .

To ensure message integrity, the sender will generate a 16-bit CCITT CRC checksum for the message, append the two byte checksum to the end of the message.

During transmission, the SENDER will segment the whole message with checksum into one or more X.25 data packets. The packet size is determined at CALL time
25 following X.25 protocol standard. The last packet of the message will not have the M(ore) flag asserted. There is no need to define either a Start Of Message (SOM) or an End Of Message (EOM) character sequence.

3. WAIT_ACK State

30 After sending a message, the SENDER will wait for the acknowledgment from the RECEIVER.

The SENDER will increment the sequence number and remove the current message from the Output Queue of the Queue Manager only if a good acknowledgment is received.

5 Depending on the error code in the acknowledgment note from the SWIFT NETWORK INTERFACE, the SENDER will log event to the Event Queue for whatever happened.

The outbound X.25 link server may manage the link with a finite state system. The state diagram is shown in Figure 9.

The states defined in this diagram are described below.

10 To Be Connected State, 301: This is the initial state. The circuit is disconnected. When link server is in this state, it's about to issue a call request to SWIFT NETWORK INTERFACE . After x25call() is issued, it goes to Pending Connect State.

15 Pending Connect State, 303: A call request was issued, the link server waits for the result. If result is OK, it goes to Connected State, otherwise, it goes to Fail Connect State. If no result from SWIFT NETWORK INTERFACE after time-out period, link server will cancel the pending call with x25cancel() and go to Fail Connect State.

20 Fail Connect State, 305: System is in this state if the pending call request failed or timed-out, and the circuit is not up. It waits on this state for a certain period of time. After the timeout period, it goes to To Be Connected State and try to connect again.

25 Connected State, 307: This is the idle state for normal operation cycle. Circuit is up, the link server is waiting for new message from applications. There is no message exchange activity going on the circuit. If message comes, it'll send it with x25send() and go to Pending Send State.

30 Pending Send State, 309: A message was sent out, the link server waits for the result. If result is OK, it goes to Wait Reply State, otherwise, it goes to Fail Send State. If no result from SWIFT NETWORK INTERFACE after time-out period, link server will cancel the pending call with x25cancel(), and go to Fail Send State.

Fail Send State, 311: The last message sending request failed. It usually happens because SWIFT NETWORK INTERFACE has already cleared the circuit

unexpectedly due to the problem found during the last message exchange, or for some other reasons. The result of the last x25send() call tells us if the circuit is up or down. If the circuit is still up, the link server should go to Reset State and clear the circuit with x25hangup() there, then reestablish the X.25 session. If the circuit is already
5 down, the link server should go to To Be Connected State immediately and reissue X25call() there.

Wait Reply State, 313: A message was successfully sent out. The link server issues x25recv() for the reply message from SWIFT NETWORK INTERFACE , then go to Pending Reply State.

10 Pending Reply State, 315: The link server waits for the result of getting reply message from SWIFT NETWORK INTERFACE . If result is OK, it goes to Process Reply State, otherwise, it goes to Fail Reply State. If no result from SWIFT NETWORK INTERFACE after time-out period, link server will cancel the pending call with x25cancel(), and go to Fail Reply State.

15 Fail Reply State, 317: The link server failed to receive the reply message from SWIFT NETWORK INTERFACE . It usually happens because SWIFT NETWORK INTERFACE has already cleared the circuit unexpectedly because of the problem found in the last message we sent to them, or for some other reasons. The result of the last x25recv() call tells us if the circuit is up or down. If the circuit is still up, the link
20 server should go to Reset State and clear the circuit with x25hangup() there, then reestablish the X.25 session. If the circuit is already down, the link server should go to To Be Connected State immediately and reissue x25call() there.

Process Reply State, 319: In this state the link server examines the reply message it Just got from SWIFT NETWORK INTERFACE related to the last
25 message sending. If the reply message indicates that the message sending is OK, the link server will do the appropriate processing like increment the sequence number, and go to Connected State, where it's ready to send the next message. If the reply message indicates there is a non-fatal protocol problem, the link server will back out the last message sending, make appropriate correction, and go to Connected State to
30 resend the message. If problem is fatal, the link server will go to Clear State and hang-up the circuit there.

Reset State, 321: The X.25 circuit is up, but there is a problem that the circuit must be cleared and reestablished. So the link server will hang-up the circuit. then go to To Be Connected State to reissue x25call().

5 Clear State, 323: Severe problem has occurred and the link server should terminate. So it will clear the X.25 circuit with x25hangup() call, then go to Shutdown State.

Shutdown State, 325: The circuit is disconnected and there is no reason for the link server to stay alive. This usually happens when severe errors happened to the circuit, or after user closed the circuit.

10 Message processing may proceed as follows. For outbound link, after the X.25 connection is up, the link is at connected state, with nomessage exchange activities active. The link thread is at its initial state:

It waits for ANY semaphore to signal. It knows which semaphores to look at.

Once a semaphore is received, the link thread does the following:

15 Make semaphore not signaled.

Retrieve but do not destroy the first message from the queue bound to the semaphore. It reads the message, and remember the application outbound sequence number.

20 Check source and destination code on the message. If not recognized, nak the message with the right application number. Signal the reply semaphore.

Put GCN code and SWIFT NETWORK INTERFACE outbound sequence number onto the message.

Append CRC-CCITT checksum of the whole message to the end.

Then send the message with x25send().

25 At this point, if the x25send() fails, or it fail to receive a CMX level reply message from the network the outbound link server treats the message sending as having failed and will execute the following procedure:

Release the semaphore, increment it by 1 so it's back to signaled state so that the environment is configured like the outbound message has never been touched.

30 Reset the outbound circuit, which means close it and reconnect. If the connection is already closed, issue a x25call() to reconnect it. Log this event.

If a CMX level reply message comes in, based on what it is, the outbound link server will respond like this:

1. Reply message is "00 - ACCEPTED". Outbound link server takes action 0:
5 Increment SWIFT NETWORK INTERFACE outbound sequence number, so next message will use a higher number. The number used after 9999 must be 000 1. Remove the message from the queue bound to the semaphore. Insert an ack message to the reply queue back to application. The ack message contains application sequence number. MQ Series will signal the reply semaphore.
- 10 2. Reply message indicates that outbound link server should take action 1 or 2. This is generally an application error, so link server should reply to application with a nak message. Remove the message from the queue bound to the semaphore. Insert an nak message to the reply queue back to application. The nak message contains application sequence number. MQ Series will signal the reply semaphore.
15 The content of the nak message is important. It should provide enough information for the application to correct the message. At least it must have severity code, error type code, and error message. Action 2 is more severe. It means the problem is at system level, not message level. The application probably should be terminated to avoid further damage and then investigate the error. The outbound link server does not
20 increment SWIFT NETWORK INTERFACE outbound sequence number, so next message will use the same number.
3. Reply message indicates that outbound link server should take action 3. This is an operational error, the link server should make correction and resend the message instead of replying to application with a nak message. Release the
25 semaphore, increment it by 1 so it's back to signaled state so that the environment is configured like the outbound message has never been touched. Take appropriate actions. e.g. If it's "32 - LOW SEQUENCE NUMBER", the link server must increment its SWIFT NETWORK INTERFACE sequence number. If it's " 13 - NO SEQUENCE NUMBER", the link server must check the reason. If it's "30 INVALID
30 CHKSUM", the link server must change the CRC polynomial and regenerate its CRC table. Revert back to initial connected state and wait for semaphore.

4. Reply message indicates that an unlikely error has occurred on the message. This usually means it's a system or X.25 problem, not with the message. The link server should keep the message in unsent status, then terminate the system and investigate the error. Release the semaphore, increment it by 1 so it's back to signaled state so that the environment is configured like the outbound message has never been touched. The application should be terminated to avoid further damage and then investigate the error.

The following table provides a brief summary of the foregoing discussion:

| Action | 0v | 1v | 2 | 3v | Undefined v |
|--|-----|----------------------|----------------------|--------------------|----------------------|
| Put Message back unsent | | | | yes, PDE next time | yes |
| send Ack to application & increment Seq. No. | yes | | | | |
| send Nak to application | | yes | yes | | |
| make other corrections | | maybe, no action now | maybe, no action now | yes | maybe, no action now |
| X.25 server terminate | | | yes | | yes |

In any event, it is preferable that the link server must log the message transfer result.

Once back to the connected state with no message exchange activity, it's time for the outbound link server to check external event to see if operator wants to close it. If it does, it'll follow the procedure described above in the general section to do so.

Exceptions to the foregoing message transfer actions are generally caused by an MQ Series problem. If the Message Queuing System does not function normally the X.25 link server will have trouble working on its way.

The major interaction points between X.25 outbound link server and message queue are listed below:

Startup point: connect to message queuing system (MQ Manager), open semaphore. This is described above.

Wait for new message from application: If MQ Manager's trigger monitor is down at this time and a new message comes in, link server keeps waiting, but unable to get the message. If MQ itself is down, link server keeps waiting. When MQ back up again, link server is still unable to get the message because it's not triggered by semaphore. Preferably the trigger monitor is synchronized with new messages in the queue.

If for some reason, link server get triggered, but unable to get the new message, it won't be able to send that message out. Instead it goes back to wait for the new semaphore triggering event.

It's possible that message is sent out, but link server fail to give sent status back to application. If this happens, when the link server got Ack/Nak from SWIFT NETWORK INTERFACE later, it destroys the original message from the message queue, but won't be able to update message status again. So application don't know the correct status of this message. In other words, it is preferably that a message and its status must be synchronized at all time.

All related operations must be bound together as atomic operation.

The Inbound Service Module takes care of the RECEIVE CHANNEL. Basically it's in one of three states:

1. LISTEN State: When the Inbound Service Module starts, it's in this states by default. It listens to incoming call request from the specified remote SWIFT NETWORK INTERFACE DTE. For security purposes, this RECEIVER will check and verify the incoming DTE, only calls from predefined SWIFT NETWORK INTERFACE DTE are accepted.
2. WAIT-MSG State: The RECEIVER will continuously receive data packets until detecting a packet without a M(ore) flag. During the process, it assembles the message by concatenating received packet data. After the whole message is concatenated together, the last two bytes is the 16-bit CCITT CRC checksum of the whole message. The real message is two bytes short from what received. So the RECEIVER will recalculate the checksum and verify with the received checksum. If the checksums don't match, the RECEIVER will send the nak

message to SWIFT NETWORK INTERFACE with the error code "30 - INVALID CHKSUM" as data.

If the checksum is good, it's removed from the tail of the message, and the RECEIVER goes to SEND_ACK state.

5 3. SEND ACK state: The RECEIVER does the following to process the message:

 The real message without checksum is queued to the Input Queue of the Queue Manager. If this step failed, the RECEIVER must safestore the message and shut itself down.

10 Send Ack/Nak back to SWIFT NETWORK INTERFACE based on the processing result of step 1.

 Log event to the Event Queue.

 The inbound X.25 link server may manage the link with a finite state system. The state diagram is shown in Figure 10. The states defined in Figure 10 are described below.

15 To Be Connected State, 401: This is the initial state. The circuit is disconnected. When link server is in this state, it's about to issue a listen from SWIFT NETWORK INTERFACE . After x25listen() is issued, it goes to Pending Connect State.

20 Pending Connect State, 403: A listen was issued, the link server waits for an incoming call from CITISWITCH. If a call comes in, and the circuit setup OK, it goes to Connected State. If the listen failed, it goes to Fail Connect State. If no result from SWIFT NETWORK INTERFACE after time-out period, link server will cancel the pending listen with x25cancel(), and go to Fail Connect State.

25 Fail Connect State, 405: System is in this state if the pending listen failed or timed-out, and the circuit is not up. It waits on this state for a certain period of time. After the time-out period, it goes to To Be Connected State and try to listen again.

 Connected State, 407: This is the starting point for normal operation cycle. Circuit is up, the link server issues x25recv() on the circuit to wait for new message from SWIFT NETWORK INTERFACE . Then it goes to Pending Receive State.

30 Pending Receive State, 409: If message comes, the x25recv() will return OK, and the link server goes to Wait Reply State to prepare for the reply message to

SWIFT NETWORK INTERFACE . If the x25recv() call failed, it goes to Fail Receive State. If no message from SWIFT NETWORK INTERFACE , the link server will stay in this state.

5 Fail Receive State, 411: The last receiving message call failed. It usually happens because SWIFT NETWORK INTERFACE has already cleared the circuit unexpectedly due to the problem found during the last message exchange, or for some other reasons. The result of the x.25recv() call tells us if the circuit is up or down. If the circuit is still up, the link server should go to Reset State and clear the circuit with x25hangup() there, then reestablish the X.25 session. If the circuit is already down,
10 the link server should go to To Be Connected State immediately and reissue x25listen() there.

Wait Reply State, 413: A message was successfully received. The link server processes the message, sends to application. Based on its result, the link server prepares the reply message and sends it back to SWIFT NETWORK INTERFACE
15 with x25send(), then it goes to Pending Reply State.

Pending Reply State, 415: The link server waits for the result of sending reply message to SWIFT NETWORK INTERFACE . If result is OK, it goes to Connected State and deals with next message. Otherwise, it goes to Fail Reply State. If the reply sending process has no result after time-out period, link server will cancel the pending
20 send with x25cancel(), and go to Fail Reply State.

Fail Reply State, 417: The link server failed to send the reply message to SWIFT NETWORK INTERFACE. It usually happens because SWIFT NETWORK INTERFACE has already cleared the circuit unexpectedly because of the problem found in the last message we sent to them, or for some other reasons. The result of the
25 last x25send() call tells us if the circuit is up or down, If the circuit is still up, the link server should go to Reset State and clear the circuit with x25hangup() there, then reestablish the X.25 session. If the circuit is already down, the link server should go to To Be Connected State immediately and reissue x25listen() there.

Reset State, 419: The X.25 circuit is up, but there is a problem that the circuit
30 must be cleared and reestablished. So the link server will hang-up the circuit with x_25hangup(), then go to To Be Connected State to reissue x25listen(),

Shutdown State, 421: The circuit is disconnected and there is no reason for the link server to stay alive. This usually happens when severe errors happened to the circuit, or after user closed the circuit.

By way of example, a typical message processing scenario may look as follows. For inbound link, after the X.25 connection is up, the link is at connected state, with no message exchange activities active. The link thread is at its initial state:

It issues x25recv() to wait for a message from SWIFT NETWORK INTERFACE .

If the x25recv() failed:

Reset the inbound circuit. If the VC is already closed, issue another x25listen() to reconnect it. The goal is to go back to the initial connected state, then try to receive another message.

Once a message is successfully received, the link server performs the message checking steps. If it encounters any error during the following checking, the link server takes actions by itself and sends CMX level reply message back to SWIFT NETWORK INTERFACE .

Check CRC-CCITT checksum of the received message. If wrong, reply "30 - INVALID CHKSUM". Then strip the checksum from the message.

Check GCN code, and the incoming SWIFT NETWORK INTERFACE sequence number. If not expected, reply " 13 - NO SEQUENCE NUMBER" or "32 - LOW SEQUENCE NUMBER". Then strip the first line from the message.

Check source address code. If it's not SWIFT NETWORK INTERFACE , reply error messages " 19 INVALID ORIGINATOR".

Check destination address code. If the link server doesn't know where to queue this message, it replies " 14 - INVALID ADDRESS".

Check message format. Various format errors may arise. Construct reply message accordingly.

It is not required that the link server notify the applications about these errors.

If the message is correct, the link server will do this:

Add application sequence number to the message.

Send the message directly to the appropriate application queue. No semaphores are involved.

Send reply message "00 - ACCEPTED" back to SWIFT NETWORK INTERFACE . If the reply is sent back successfully, increment SWIFT NETWORK INTERFACE inbound sequence number.

Increment application inbound sequence number.

5 In case the link server fails to send any reply message back to SWIFT NETWORK INTERFACE, whether its ack or nak, the link server must reset the inbound circuit to its initial connected state and wait for new messages coming from SWIFT NETWORK INTERFACE .

10 Most failures in the inbound message process will result from problems with the MQ Series engine. The major interaction points between X.25 inbound link server and the message queue are listed below:

Startup point: connect to MQ Manager, open semaphore. This is described above.

15 The message received from the SWIFT NETWORK INTERFACE may be either a real transaction message from SWIFT or the Nak message from SWAN or SWIFT, mostly due to message format problems. If it's a Nak from SWAN or SWIFT, the link server will put the Nak message onto Error Queue instead of normal application message queues. If it fail to put it onto Error Queue, the link server will reply "27 - UNDELIVERABLE MESSAGE" back to SWIFT NETWORK
20 INTERFACE .

If the link server fails to parse the incoming message from the SWIFT NETWORK INTERFACE, or fail to put it into application message queue due to MQ problems, it'll send a "27 UNDELIVERABLE MESSAGE" back. The SWIFT NETWORK INTERFACE will shutdown the circuit, then try to connect again. When
25 the link server finds the circuit is down when it tries to get the next message, it'll listen again to setup connection, then receive the next message.

Sometimes after the link server sent a message to application, it can not get the status message indicating that the message has been delivered to its application system. This may because that message queueing service is down, or message agent
30 server RPC server program is not up. If this happened, the link server is unable to send reply back to the SWIFT NETWORK INTERFACE . The CMX protocol may hang up in this dead lock.

Sometimes the link server may fail to send reply back to the SWIFT NETWORK INTERFACE because circuit is already down for some reason, most probably the SWIFT NETWORK INTERFACE cleared it. In this case the link server will listen again, but after reestablished the circuit it won't resend the reply. Instead it'll wait for the next incoming message, most likely is the previous one with PDE. If this is the case, it should discard the new message and reply "00 - ACCEPTED".

Preferably the message agent server is constructed so as to provide synchronization between the message agent server RPC server (MQ-Application Interface) and the X.25 server (Message Queue Network Interface). Synchronization is desirable to achieve the following goals: no message from the message agent server RPC server to the X.25 server is lost; no message from the message agent server RPC server to the X.25 server is lost; no message is sent more than once from the message agent server RPC server to the X.25 server; no message is sent more than once from the X.25 server to the message agent server RPC server.

Synchronization may be accomplished in a variety of manners. One technique is described below.

The message agent server RPC server and the X.25 server communicate through semaphores, namely SEM_OUT_CTSW, SEM_IN_FXO, SEM_IN_FXM, and SEM_IN_STM. The message agent server RPC server has to make sure that the value of the semaphore SEM_OUT_CTSW always equals to the number of messages in the queue QL_OUT_CTSW. The X.25 server has to make sure that the value of SEM_IN_FXO (SEM_IN_FXM, SEM_IN_STM, respectively) always equals to the number of messages in the queue QL_IN_FXO (QL_IN_FXM, QL_IN_STM, respectively).

On a Windows NT platform, after a semaphore is created, it is persistent until all the processes that have accessed it are terminated. Therefore, after the message agent server RPC server creates SEM_OUT_CTSW, it will be persistent until both message agent server RPC server and the X.25 server are terminated. Same goes with SEM_IN_FXO, SEM_IN_FXM, and SEM_IN_STM.

The following table details possible failure scenarios and methods for recovering from the failures.

| | Detection | Scenario | Action |
|--|--|---|--|
| MAS RPC Server Recovers | attempt to open SEM_OUT_CTSW succeeds | X.25 server has been up when MAS RPC server was down | No special action needed |
| MAS RPC Server Recovers | attempt to open SEM_OUT_CTSW fails | X.25 server is down or was down but has come back up | create SEM_OUT_CTSW with an initial value equal to the number of messages in QL_OUT_CTSW |
| X.25 Server Recovers | attempt to open SEM_IN_FXO succeeds (similar for SEM_IN_FXM, SEM_IN_STM) | MAS RPC server has been up when X. 25 server was down | No special action needed |
| X.25 Server Recovers | attempt to open SEM_IN_FXO fails (similar for SEM_IN_FXM, SEM_IN_STM) | MAS RPC server is down or was down but has come back up | create SEM_IN_FXO with an initial count equal to the number of messages in QL_IN_FXO_ (similar for SEM_IN_FXM, SEM_IN_STM) |
| MasReceiveMsg Is called (in MAS RPC Server) | attempt to open SEM_IN_FXO fails (similar for SEM_IN_FXM, SEM_IN_STM) | x.25 server is down | create SEM_IN_FXO with an initial count equal to the number of messages in QL_IN_FXO_ (similar for SEM_IN_FXM, SEM_IN_STM) |
| X.25 Server Waits to Be Signaled to Send a Message | attempt to open SEM_OUT_CTSW fails | MAS RPC server is down | create SEM_OUT_CTSW with an initial value equal to the number of messages in QL_OUT_CTSW |

These failures, and recovery therefrom, are discussed in more detail below. When X.25 Server recovers it tries to open relevant semaphores and MAS QM.

Outbound Service: Open SEM_ERROR_MAS_SEM_OUT_CTSW.

5 Failed: MAS RPC Server is down. Unable to get new messages, but there may be old messages in the queue. So create these semaphores and initialize to the message count in the relevant queue.

 Succeeded: MAS QM is up. So send all messages *in QL_OUT_CTSW, then wait.

10 Failed: MAS QM is down. No way to retrieve any message. So quit.

 Succeeded: MAS RPC Server is up. Semaphore must be initialized to the message count. Then it connects to message agent server QM.

 Succeeded: MAS QM is up. So send all messages in QL_OUT_CTSW, then wait.

15 Failed: MAS QM is down. No way to retrieve any message. So quit.

 Inbound Service: Open SEM_ERROR_MAS_all semaphores for application queues.

20 Failed: MAS RPC Server is down, but we can still send messages to the queues. So create these semaphores and initialize to the message count in the relevant queue.

 Succeeded: MAS QM is up. So start operation.

 Failed: MAS QM is down. No way to send message over. So quit.

 Succeeded: MAS RPC Server is up. Semaphore must already be initialized to the message count. Now connects to MAS QM.

25 Succeeded: MAS QM is up. So start operation.

 Failed: MAS QM is down. No way to send message over. So quit.

 When message agent server RPC Server recovers, it tries to open SEM_ERROR_MAS and SEM_OUT_CTSW.

30 Failed: X.25 Server is down. Although the messages can't be sent to the SWIFT NETWORK INTERFACE, this server should still send message to MQ. So create these semaphores and initialize to the message count in the relevant queue.

Succeeded: MAS QM is up. So send all messages to QL_OUT_CTSW, then wait.

Failed: MAS QM is down. No way to send message. So quit.

5 Succeeded: X.25 Server is up. Semaphore must be initialized to the message count. Then it connects to message agent server QM.

Succeeded: MAS QM is up. So send all messages to QL_OUT_CTSW, then wait.

Failed: MAS QM is down. No way to send message. So quit.

10 In the message agent server RPC Server, MasReceiveMsg Is called to retrieve messages from message agent server application queues. So it attempts to open SEM_ERROR_MAS & all semaphores for application queues.

Failed: X.25 Server is down. Unable to get new messages, but there may be old messages in the queue. So create these semaphores and initialize to the message count in the relevant queue.

15 Succeeded: MAS QM is up. So start operation.

Failed: MAS QM is down. No way to get message over. So quit.

Succeeded: X.25 Server is up. Semaphore must already be initialized to the message count. Now connects to MAS QM.

Succeeded: MAS QM is up. So start operation.

20 Failed: MAS QM is down. No way to get message over. So quit.

One special situation may occur when a message agent server QM Triggrrer Monitor dies and restarts, while the X.25 Server is always up so the semaphore is never deleted. Resynchronization may effected by restarting the servers.

25 An alternative embodiment of a computing environment utilizing the system of the present invention is depicted in Figure 11. The depicted embodiment includes 4 servers, a message agent server (MAS), 600 for interfacing with application programs 601; an X.25 Server, 602, utilized for access to the SWIFT network, 603; an administrative server, 604 which provides a human interface to the system, through
30 graphical user interface 605; and a File Tranfer Server (FTS), 608 for communicating with other customers, 609. The MAS, X.25 and Administrative servers have access to

the message queue, 610 which is preferably an IBM MQ Series. The FTS server's access to the message queue is through the MAS server.

The X.25 server is responsible for communicating with a SWIFT network gateway and ultimately the SWIFT network through an X.25 protocol. This protocol maps a logical connection to a station ID as a means of identification and the sequence number as a means of synchronization. Each side needs to keep track of its sequence number, only messages with the same expected sequence number are accepted. Details relating to the X.25 protocol and message traffic to and from the SWIFT network are set forth in the preceding sections.

The FTSIN server is responsible for processing files received from customers. Customers may use a file transfer facility called "Connect:Direct" from Sterling Commerce, Inc. to send files to the server. Upon detecting the arrival of a file, FTSIN breaks the contents of the file into records, optionally reformats and submits the record to MAS as a message or writes the reformatted record to a file for delivery to an application.

The FTSOUT server is responsible for processing files or messages to customers. An application can use Connect:Direct to transfer files to the FTSOUT server. FTSOUT will reformat the file according to the customer specification and leave the file in a specific directory to be downloaded by the customer. An application can also use the messaging infrastructure to send messages to FTSOUT and FTSOUT will treat the message as a file from the application.

The Administrator Graphical User Interface (GUI) provides the management tools for MAS. Through MAS GUI, an administrator is able to monitor the health of the system and to control and manage the system components. It also provides facilities to query the messaging metrics, and the event database.

The administrative server is an interface between the Administrator clients and the messaging infrastructure. This server allows an administrator to start and to stop a system component, to examine the message queues and to move messages from one queue to another queue.

Figure 12 depicts a MAS data access model. An Event Log Server, 620 may be responsible for recording all events in a local machine and to upload the events at a regular interval to a common database, 630. Even if the database is not available or

slow, the server still writes the events to a local log file as a backup and also to provide a consistent level of performance. The event log server is also programmed to purge the database and the local log files after a defined retention period.

5 The MAS may further include a database access server, 622. The database server provides a common interface to the database, 630. A MAS process may use this server to access the database. The database may store the MAS events, the sequence numbers, the daily volume counts, the metrics information and the FTS customer profile. The database may be constructed in the manner described above with reference to other embodiments of the present invention.

10 MAS may use DCE security for user authentication. Only users belonging to authorized groups can gain access to MAS services. Following is the convention to enforce DCE security.

The principal who starts a server will have the name defaulted to %host%/%service% where %host% is the machine name and %service% is the
15 service name.

A server principal should belong to the group "MASServers."

A human MAS client should belong to the group "MASUsers."

A machine MAS client should belong to the group "MASC callers."

A MAS service is generally authorized for the groups "MASServers,"
20 "MASC callers" and "MASUsers."

In addition to DCE security, MAS servers may also be subject to strict NT security. MAS servers run under an NT account "MASServices." This account has full permission to access files under the MAS root directory.

25 Message queue objects are protected by MQ authorization. Only authorized clients can gain access to an MQ object. A special NT group "MQClients" is defined for this purpose. An NT user can access MQ queues if the user belongs to the "MQClients" group. For this reason, the NT "MASServices" account also belongs to the "MQClients" group to allow MAS servers to access MQ queues.

30 As described above, an embodiment of the system of the present invention may include an administrative interface to allow an administrator (i.e. a human) to monitor, control, manage and investigate the message agent server system. The administrative client may be use ActiveX controls installed as follows.

To ensure proper operation of the client program, ActiveX controls need to be installed in the client machine. The following ActiveX OCX files should be included:

- i. Graph32.OCX
- ii. Grid32.OCX
- 5 iii. MSCal.OCX
- iv. Tabct132.OCX

The Administrative Graphical Interface (GUI) may utilize the following operational procedures to monitor, control, manage and investigate the MAS system. Preferably the system will be configured so that a product administrator must use a
10 suitable data encryption program, such as Smartgate, and DCE authorization to login to MAS.

After the user is authenticated, the MASGUI program will display a menu screen, Figure 13, from which the administrator can select:

- Monitoring for system monitor.
- 15 Services for service startup and shutdown.
- Messages for message handling.
- Events for event searching.
- Functions for special functions such as changing password and purging status queue.
- 20 Metrics for message statistics and
- FTS for managing the FTS customer profile.

The MAS system may be monitored utilizing a system monitor screen. The system monitor screen, Figure 14, displays the current status of all MAS servers on all machines, the number of messages on application queues, the current sequence
25 number of each Citiswitch link and the daily traffic volume on the link.

When an alert shows up on a system monitor, Figure 15, a product administrator may immediately examine its contents and acknowledge that the alert has been handled. If the alert relates to a bad message, the administrator may extract the message from the error queue and inform the sender about the problem.

30 On the inbound direction from Citiswitch to MAS, the X.25 inbound server will adjust its sequence number to match the expected number from Citiswitch. An alert is generated to alarm the condition but the server still continues to run.

On the outbound direction to Citiswitch, the X.25 outbound server will adjust its sequence number to the next higher number or to the next lower number depending on the Citiswitch error code. However, this adjustment is subject to a gap, which is configured to 3, to limit the number of retries. If the sequence gap is larger than this
5 configured parameter, the server will terminate. An administrator should then request Citiswitch to change its next expected sequence number to synchronize with that of X.25 server. Note that the monitor screen, Figure 16, shows the current sequence number, the next expected number is one higher than the current number. For example, if the monitor screen shows "1234" for the station GWS, an administrator
10 should request Citiswitch to change its sequence number to "1235."

The administrator may control the message agent server system as follows.

On the administrator GUI, Figure 17, select the services tab, choose the machine and take "All Services" option. Hit the "Stop" button to shutdown all MAS services on that machine.

15 To shut down a single service, an administrator can either select the service from the services tab or double click on an active service in the monitor screen. This will stop a single service. The MASEvtLog service is a critical component that other services depend on. Never shut down the MASEvtLog service.

To start up a service, an administrator can either select the service from the
20 services tab in the GUI or double click on an inactive service in the monitor screen. This will start a single service.

A system reboot generally will start all service configured on a machine. From the GUI, an administrator can select the services tab, choose the machine and take "All Services" option. Hit the "Start" button to start all MAS services on that
25 machine.

The administrator may manage the message queues as follows.

Message queues may be browsed as follows. On the monitor screen, double click on a queue to start another window. In this window an administrator can select an option to browse, to delete or to move the message. The message text, the message
30 ID and the queuing time as displayed on the window, Figure 18. A message on an application queue can only be moved to the error queue.

Error queues may be handled as follows. On the monitor screen, double click on the QL_MAS_ERROR queue to open another window. Select the browse option to get the first message on the error queue. The reject reason is displayed at the bottom of the message, Figure 19. An administrator should inform the sender why the message was rejected, a note of explanation should be entered before the message can be deleted from the error queue.

The administrator may also perform research and investigations on the functioning of the message agent server.

Event searching may be performed as follows. On the monitor screen, select the events tab to open a new window, Figure 20, in which an administrator can enter parameters for querying the events database. The event and its parameters are fully described in Appendix C:MAS Events. To execute the query in the most efficient way, as many search parameters must be specified as possible, particularly the time span parameter and the severity parameter. Note that the parameters are case sensitive.

The administrator may also perform metrics queries.

Message volume statistics may be searched as follows. The message volume statistics screen, Figure 21 allows an administrator display the hourly or the daily traffic profile within a certain period for the inbound direction from Citiswitch or the outbound direction Citiswitch. The data can be displayed in the form of a table, a line chart, a two-dimensional chart or a three-dimensional chart. The chart may also be printed from a printer.

Message Direction: The message direction is relative to MAS system. Inbound means the message flows from Citiswitch to MAS. Outbound means the message flows from MAS to Citiswitch.

Timeline: The message volume statistics can be shown hourly or daily. For example, if you want to see daily message statistics from 05/01/98 to 05/10/98. The time interval 0 starts from 05/01/98 and end at time interval 10 which is 05/10/98. The same idea applies to statistics data shown hourly.

Starting Date: The starting date begins from the midnight of your specified date, e.g., 05/05/98 means 05/05/98 00:00:00.

Ending Date: The ending date ends at the midnight of the next day of your specified date, e.g., 05/05/98 means 05/05/98 23:59:59.

Failed Messages and Message Delay Statistics may be displayed as failed message statistics, Figure 22, over a certain period categorized by application or by failure reason. The message delay time is the difference between the dequeuing time and the queuing time of a message. The message delay window provides the worst delay time and the delay time of total percentage messages according to normal distribution model. The formula used to calculate the value of the delay time is: $P = \text{Mean} + (Z\text{-Score} * \text{Standard Deviation})$. Detail of this formula may be found in any statistics book in the chapter of normal distribution.

Downtime statistics may be displayed as the down time statistics of MAS services and their respective components over a certain period of time. The entry is based on host name, service name, and component name (only for external component). The service downtime shows the total downtime of a particular service on a particular host. i.e. If the same service is running in two different hosts, and when they go down, two separate entries will be recorded in the table. The component downtime shows the total downtime of a particular component on a particular host with the same service name.

Advanced message statistics provides the message statistics from different views, Figure 23: by message types, by BIC codes or by GCN codes.

The administrator may also monitor FTS customer profiles.

The Profile Selection Screen allows for selection of the profiles to view/update, Figure 24.

The "Inbound Profiles" Screen is accessible through the "Inbound Profiles" button on the profile selection screen. The inbound profiles dialog box, Figure 25, displays a list with all inbound profile records in the database. The "Cust. ID," "Pattern #" and "Pattern" fields are displayed, as shown in the sample below. A description of the buttons (and their use) found in this screen is as follows:

New is used to add a new inbound profile record in the database.

New Pattern is used to add a new pattern.

Delete: is used to remove the selected entry from the database. Multiple entries can be selected use the Ctrl key; a range of contiguous records can be selected with the Shift key.

5 Details is used to show details about the selected entry. Double clicking on an entry has the same effect. If multiple records are currently selected, then this button is disabled. A sample of the “Inbound profile details” screen is shown below.

Close is used to dismiss the dialog box and return to the previous screen.

The “Inbound Profile Details” Screen dialog box, Figure 26, displays all the fields in the selected inbound profile record:

10 Cust ID is an (up to) 11-character identification string assigned to this customer. It is a case-sensitive field. A subdirectory having the same name is created off the root of the client directory defined by the environment variable FTS_CLN_ROOT.

15 Pattern # is a sequence number assigned to each pattern defined for the “Cust ID” above. The pair (Cust ID, Pattern #) uniquely identifies a pattern for “Cust ID” in the database. Pattern matching with a file name is attempted in the increasing order of “Pattern #”: only the first match, if any, is reported.

20 Pattern is the actual string used for pattern matching with a file name. The only wild characters allowed are ‘*’ (any string, including the empty string), and ‘?’ (any single character). Limited to 64 characters.

25 Translate id is the numeric identifier of the translation method used to process the records (using MFL calls) in files matching the pattern string above. If this field is 0, then no parsing is performed (passthrough message). This is the only field currently used for message parsing with MFL.

30 Parse id is the numeric identifier of the parsing method to use to process the records (using MFL calls) in files matching the pattern string. Not used currently.

Format id is the numeric identifier of the formatting method to use to process the records (using MFL calls) in files matching the pattern string.

5 Start record is a special string marking the start of a record in the file. It's used together with the "End record" string below to isolate individual records in the file.

10 End record is a special string marking the end of a record in the file. It's used together with the "Start record" string above to isolate individual records in the file. Special characters (carriage-return (CR), line-feed (LF)) are entered in a C-like fashion (CR: '\r,' LF: '\n'). Both "Start record" and "End record" strings are limited to 11 characters. For example, the '-' character alone on a line will be entered as "-\r\n" in any of these two fields.

15 Destination specifies the final destination of the record(s) in the file. There are two mutually exclusive options, displayed as radio buttons:

20 File: the records are assembled in a file that is placed in the directory specified in the "Dest. directory" field (limited to 255 characters). The string in the "Dest. extension" field (at most 3 characters), if not empty, is appended to the file name before placing it in the directory.

25 MAS: the records in the file are queued up to MAS for delivery to the final destination. The target application ID is specified in the "Dest. id" field. It is either a numeric identifier (a single digit) or a 5-character string. Each record in the file is sent as a single MAS message.

Several actions are available, identified by the buttons on the right-hand side:

30 Previous Pattern is used to display details about the previous record (in the "Pattern #" order) of "Cust. ID" in the database. An error message is displayed if there is no previous record to display.

Next Pattern is used to display details about the next record (in the "Pattern #" order) of "Cust. ID" in the database. An error message is displayed if there is no next record to display.

5 Insert is used to add a new record to the database. It is enabled only if any of the "Cust. ID" and "Pattern #" fields is modified.

Update is used to save the changes in the current record to the database. It is enabled if any field but "Cust. ID" and "Pattern #" is modified.

10

Delete is used to remove the current record from the database (and from the list in the "Inbound profiles" screen). This action dismisses the dialog box and returns to the "Inbound profiles" screen.

15 Exit is used to leave the dialog box without saving any changes. Control is passed to the "Inbound profiles" dialog box.

The foregoing administrative functions are provided by way of example. Additional administrative functions may be incorporated into the message agent server depending of the needs of the computing environment.

20

As discussed above, the present invention also provides a message format library which may be separate from the message agent server.

25

According to the present invention a message format library may include one or more of the following functions: means for parsing messages; means for translating messages; and means for validating message formats. The means for validating may utilize the means for parsing such that the message is divided into its components by the means for parsing so the means for validating reads the components and compares them to a database of allowed components. In similar fashion the means for translating may utilize the means for parsing such that the message is divided into its components by the means for parsing so the means for translating may translate the components into a different format based upon a database of acceptable different format components. By way of example, the means for parsing may derive a

30

destination address within a message and/or a source address within a message. The translating means may translate messages in a first format into a second format and translate messages in the second format into the first format.

In another aspect the present invention provides a message format library
5 comprising:

means for parsing messages;

means for formatting messages into defined message formats;

means for translating messages from one message format to another message
format. The capabilities of the message format library are provided through a set of
10 application programming interfaces which are accessible to applications as a static
library or dynamic link library. In a preferred embodiment, the message format
library further comprises a user interface in GUI (graphical user interface) format
which prompts a user to enter message data. The user interface may also be part of
the static library or dynamic link library. The message format library is particularly
15 advantageous for parsing, formatting and translating structured financial messages,
including SWIFT messages, ISITC messages, fixed-length and comma/tab-delimited
messages; and messages needing to be formatted as structured financial messages.

Figure 27 provides a graphic overview of an embodiment of a message format
library of the present invention. As shown in Figure 27, a message format library 500
20 may receive and send messages from application programs or networks 502. The
message format library will support a variety of different message types, such as the
message types represented in the type library 504. The messages may be broken
down into components and fields such as those represented in the translation,
matching and reconciliation engine 506. The message format library will have access
25 to databases 508 including message format and business logic data.

An embodiment of a message format library of the present invention is
described in detail below. Additional embodiments may be constructed in similar
fashion.

The message format library, or MFL, preferably uses Norm Chomsky's
30 regular expression to represent message formats. Messages are broken into
components and sub-components and their regular expression representations are
stored in a relational database. The format of a certain message type is retrieved from

the MFL database to build a finite automation, which is used to parse and format messages of that particular message type. After a message is parsed, it can then be translated to another message (or messages) of a different message type (or different message types).

5 These functionalities are abstracted into a set of application programming interfaces (APIs) wrapped in a static C library and a dynamic linked library (DLL). These libraries and the MFL format database (the meta-data) constitute the Message Format Library.

10 The MFL C interface is implemented both as a static library and a dynamic link library. The following interface functions are provided in both.

MessageParse accepts a formatted message and parses it into its constituent fields and variables;

15 MessageHeaderParse accepts a formatted message and parses only its header;

MessageFormat accepts a list of fields and variables and formats a message;

20 MessageHeaderFormat creates a formatted message header;

MessageTranslateOneToOne translate one input message into another message according to a set of translation criteria;

25 MessageTranslateOneToMany translate one input message into an array of output messages;

MakeCommonReference creates field 22 (Common Reference) in a SWIFT message;

30 GetMFLErrors retrieves errors generated by any of the above function.

LoadMessage Pre-load message formatting information for a specific message type from database to avoid spontaneous loading. This function improves run-time performance.

- 5 LoadMessageGroup Pre-load messages formatting information for a group of message types from database.

LoadMatch Pre-load match rules from database (include translation rules).

- 10 These functions are specific to the DLL:

FreeMFLString frees a character string returned by a MFL DLL function call;

- 15 FreeMFLStringArray frees a character string array returned by a MFL DLL function call;

FreeMFLIntArray frees an integer array returned by a MFL DLL function call.

- 20 The Message Format Library is available both as a static library and a dynamic link library. If you are using the library in the Windows environment, it's recommended that you use the DLL instead of the static library.

- 25 The header include file for both the DLL and the static library is named mfl.h. The 32-bit DLL using ODBC database access is called mflacc.dll and the 16-bit DLL mflacc16.dll. The 32-bit DLL using Sybase dblib data access is called mflsyb.dll. There is no 16-bit DLL using Sybase. The import libraries for the 32-bit DLL are mflacc.lib and mflsyb.lib, respectively.

- 30 When implemented on a Windows NT platform the file userpass.txt contains information needed to access the MFL database (including database server name, database name, application name, and encrypted username and password). An environment variable MFLDBDIR has to be defined for the path of the file. For example, if MFLDBDIR=C:\MFL\MBD, then the file userpass.txt should be in the directory C:\MFL\MDB.

The default userpass.txt that comes with the library is readily usable in most cases. However, this file may be changed with the program mflencpt.exe as follows:
mflencpt server_name db_name app_name user_name password output_file

Use the data source name in place of the server name if ODBC is used to access MFL database. Notice that the output file has to be "userpass.txt" to work with this version of the format library.

MFL functions may be made available on a VAX platform as a static library. The userpass.txt file is not necessary on VAX. However, a user needs to setup your environment variables correctly by logging on to the ITS environment with the following command:

```
@itscom:its_login admin mas
```

A user may also need to set the environment variable USER to be MAS_TST.

This MFL interface functions are described in detail below with reference to SWIFT messages.

The AppId is used to indicate a particular application of certain message types. Each AppID corresponds to a unique header format.

The enum MsgType can be one of the following:

SWIFT;

SWIFT_SORT;

REPORT.

The value SWIFT is usually used for this version of the DLL. If SWIFT_SORT is passed in a formatting function, MFL performs limited field sorting -- it sorts fields in non-repeatable blocks. Fields can be passed in random order except that the fields in repeatable blocks have to follow their order in their corresponding blocks. Empty string in an output variable indicates that this optional variable is missing.

For any SWIFT message, the order of elements in the parameter *variables* depends upon whether the message is tagged incoming or outgoing. Any message sent to SWIFT should contain I as the first letter after the colon in the block 2 of the message, whereas any message coming from SWIFT will contain O as the first letter, 12 characters, with the 9th character set to X. Note that incoming and outgoing is relative to SWIFT network, not the application processing the message.

| Variable[] | Variable Name | Variable Format | Default Value |
|------------|--------------------------|---|------------------------------|
| 0 | SWIFT Message Type | Exactly 3 numbers (ex. 300). | Mandatory field. No default. |
| 1* | Sender's SWIFT Address | Exactly 12 characters. | Mandatory field. No default. |
| 2* | Receiver's SWIFT Address | Exactly 12 characters, with the 9th character set to X. | Mandatory field. No default. |
| 5 | Banking Priority | Exactly 4 characters. | XXXX. |
| 6 | MUR | Up to 16 characters. | |
| 7 | Session Number | Exactly 4 numbers. | 0000. |
| 8 | Sequence Number | Exactly 6 numbers. | 000000. |
| 11 | Message Priority | 1 of the following letters: U = Urgent S = System N = Normal | N. |

A list of variable for outgoing messages is shown below.

| Variable [] | Variable Name | Variable Format | Default Value |
|-------------|--------------------------|---|------------------------------|
| 0 | SWIFT Message Type | Exactly 3 numbers (ex. 300). | Mandatory field. No default. |
| 1* | Receiver's SWIFT Address | Exactly 12 characters. | Mandatory field. No default. |
| 2* | Sender's SWIFT Address | Exactly 12 characters. | Mandatory field. No default. |
| ** | Input Time and Date | Exactly 10 characters. | Mandatory field. No default. |
| 4** | Output Time and Date | Exactly 10 characters. | Mandatory field. No default. |
| 5 | Banking Priority | Exactly 4 characters. | XXXX. |
| 6 | MUR | Up to 16 characters. | Optional field. No default. |
| 7 | Session Number | Exactly 4 numbers. | 0000. |
| 8 | Sequence Number | Exactly 6 numbers. | 000000. |
| 9** | Sender's Session Number | Exactly 4 numbers. | Mandatory field. No default. |
| 10** | Sender's ISN | Exactly 6 numbers. | Mandatory field. No default. |
| 11 | Message Priority | 1 of the following letters: U = Urgent S = System N = Normal | N. |

* SWIFT address differs from the 11 character ISO BIC code and branch code by an additional logical terminal code, which should be inserted in the 9th position of the BIC code. The logical terminal code for all messages sent to SWIFT should be X, i.e., for all messages tagged I in block 2, the 9th character of the SWIFT address should be X. Note also that the sender and receiver addresses are reversed for incoming and outgoing messages.

** To format a message with 'O' tag in block 2, variables 3, 4, 9, and 10 must be passed in.

Whenever an MFL function succeeds, it returns a positive value. Otherwise, it returns a negative value, which is the reference number for the errors encountered in the call. An application can call GetMFLErrors to retrieve these errors.

When a failure occurs, the parsing functions will look for the next good field and resume parsing while registering an error. The formatting and translation functions work in this best-effort fashion as well.

All MFL functions allocate space needed for the returned parameters. Therefore, it's not necessary for the calling program to allocate memory for the parameters beforehand. However, to prevent memory leaks, the program should free parameters returned by MFL functions at suitable times.

The MFL may include the following parsing functions.

MessageParse();

MasMsgParse();

MessageHeaderParse()

The MessageParse() specifications may be as follows:

```
long MessageParse (
    [in]  int    Aid,
    [in]  enum   MsgType,
    [in]  char*  MessageText,
    [out] int*   MsgId,
    [out] char***variables,
    [out] char*  FieldCount,
    [out] int**  FieldNumber,
    [out] char** FormatTag,
    [out] char***FieldCaption,
```

[out] char***FieldContent)

The MasMsgParse() specifications (RPC version of parsing interface) may be as follows:

```

long MasMsgParse (
5         [in]   char*  Aid,
           [in]   char*  MessageText,
           [out]  char** MsgId,
           [out]  char** FromBic,
           [out]  char** ToBic,
10          [out]  char** GenTime,
           [out]  char** RecvTime,
           [out]  char** Priority,
           [out]  char** MUR,
           [out]  char***FieldNumber,
15          [out]  char***FormatTag,
           [out]  char***FieldCaption,
           [out]  char***FieldContent)

```

The MessageHeaderParse() specifications may be as follows:

```

long MessageHeader Parse (
20         [in]   int    Aid,
           [in]   enum   MsgType,
           [in]   char*  MessageText,
           [out]  int*   MsgId,
           [out]  char***variables)

```

25 MessageParse() take a formatted swift message as input, and breaks it apart into the header variables and message body field tags and values. It returns an error status if it is unable to parse the data correctly.

 The fields are loaded into arrays FieldNumber, FormatTag, and FieldContent, with the number of fields placed into FieldCount.

30 MessageHeaderParse() takes a formatted SWIFT or CITIDEX message as input, and parses only the header of the message, returning the values MsgId,

FromBic, ToBic, GenTime, RecvTime, Priority, MUR as a character string array in the parameter variables.

The MFL may perform the following formatting functions:

MessageFormat();

5 MasMsgFormat(); and

MessageHeaderFormat().

The MessageFormat() specifications may be as follows:

```
long MessageFormat (  
    [in]  int    Aid,  
10      [in]  enum  MsgType,  
        [in]  int    MsgId,  
        [in]  char** variables,  
        [in]  int    FieldCount,  
        [in]  int*   FieldNumber,  
15      [in]  char*  FormatTag,  
        [in]  char** FieldContent,  
        [out] int*   txt_length;  
        [out] char** MessageText)
```

20 The MasMsgFormat() specifications (RPC version of formatting interface) may be as follows:

```
long MasMsgFormat (  
    [in]  char*  Aid,  
        [in]  char*  MsgId,  
25      [in]  char*  FromBic,  
        [in]  char*  ToBic,  
        [in]  char*  priority,  
        [in]  char*  MUR,  
        [in]  char*  FieldNumber,  
30      [in]  char** FormatTag,  
        [in]  char** FieldContent,  
        [out] char** MessageText)
```

The MessageHeaderFormat() specifications may be as follows:

```

long MessageHeaderFormat (
    [in]  int    Aid,
    [in]  enum   MsgType,
    [in]  int    MsgId,
    [in]  char** variables,
    [out] int*    txt_length;
    [out] char** MessageText)

```

MessageFormat() are used to validate and construct a swift message. All formatting checks are performed here. The function checks for mandatory fields, and verifies the values of all of the fields, including size. If an error is detected in formatting, the return value of the function is the Error Reference number that can be used to find the text errors corresponding to this message with a call to GetErrors().

The first parameter, Aid specifies the application Id that is requesting the function, and MsgId specifies the message id of the SWIFT message. The value of MsgType for all current messages should be SWIFT. The next string array, variables, should contain the values FromBic, ToBic, Priority, and MUR in that order.

The three arrays of string are the specific fields of the SWIFT message.

FieldNumber and FormatTag together form the field tag (i.e., 71a). The corresponding array position in FieldContent is the value of that field. MessageText is the returned formatted Swift message including header, trailer and fields. FieldCount contains the number of fields that are passed into the routine.

MessageHeaderFormat() takes the parameters Aid, MsgId, MsgType, and variables as input, and returns the formatted message in the parameter MessageText.

In addition the MFL may include a MakeCommonReference() function.

The MakeCommonReference() specifications may be as follows:

```

long MakeCommonReference (
    [in]  char*  Aid,
    [in]  char*  fromBic,
    [in]  char*  toBic,
    [in]  char*  price,

```

[out] char** commonRef)

MakeCommonReference() function takes the parameters application id, fromBic, toBic, and the value of field 36, and builds the common reference field according to SWIFT specifications. The returned value, commonRef, should be used as the value of field 22 (22C) in a subsequent call to MessageFormat (or MasFormat).

Translation functions provided by the MFL include the following:

MessageTranslateOneToOne(); and

MessageTranslateOneToMany().

The MessageTranslateOneToOne() specification may be as follows:

```

10      long MessageTranslateOneToOne (
          int    tid,
          char   **variables,
          char   *from Text,
          char   **to Text)

```

15 The MessageTranslateOneToMany() specification may be as follows:

```

      long MessageTranslateOneToMany (
          int    tid,
          char   **variables,
          char   *fromText,
20      char   ***toText,
          int    *successCount,
          int    *failureCount)

```

Both MessageTranslateOneToOne and MessageTranslateOneToMany load a set of translation criteria defined in the MAS_MATCH table and use them to translate a message (from Text) into another message or any array of messages (toText). The parameter variables, an array of strings, also may be used to specify values of parameters defined for the output text. If you do not wish to specify any parameters, simply pass in NULL.

In addition to toText, MessageTranslateOneToMany also returns the number of output messages generated in successCount. Correctly formatted messages are saved in the string array toText. MessageTranslateOneToMany will return the number of successCount if the failureCount is 0. If the failureCount is greater than 0,

the function will return an error reference number. Note that even if the function returned an error reference, some of the block may have been formatted correctly. The application should decide on the appropriate error handling strategy in this case.

The translation process may include the following three steps:

- 5 1. Parse the source message into an array of fields and an array of variables;
2. Translate these two arrays into another couple of field array and variable array;
3. Use the new array of fields and the new array of variables to format the
10 target message(s).

Both the source message format and the target message format have to be defined in the MFL meta-data, i.e., the messages that can be translated from and into are those whose formats are defined in the MFL meta-data.

For example, a SWIFT MT571 message can be a source or target message
15 because MT571 format is defined in MFL meta-data.

A comma-delimited message format may be defined “(AccountID, SecurityID, ValueDate, Quantity, Price, CrossCurrency)” in MFL meta-data, then a message in this format can be a source or a target message in a MFL translation function.

MFL translation functions support these translation capabilities:

20 Structural field and variable mapping, e.g., from MT571 field 35H to field Quantity. Any field or variable defined in the source message format can be mapped to any field or variable defined in the target message format. One field in a source message can be translated into many fields in the target message and vice-versa.

25 Value transformation, e.g., from “USD” in a MT571 field 33B to “1” for field CrossCurrency. Value transformation can make use of mapping tables provided by applications and stored in MFL database.

Both structural field and variable mapping and value transformation can be conditional. Target fields and variables can get different values based on
30 contents, existence, or absence of source fields and variables. Default values can be stored in the target message(s) if it is not composed from the source message.

Operations that can be performed on numeric data include quality test, inequality tests, and range test. Operations that can be performed on string data include variable-length sub-string extraction and comparison, acronym extraction and comparison, and word reordering. More sophisticated operations on numeric and string data can be performed by calling exit functions.

Translation rule definitions may be established as follows.

Each mapping in a translation is defined as a translation rule in MFL meta-data. A group of these rules constitute a full translation. Each group of translation rules is identified by a translation id. Translation rules are stored in the table MAS_MATCH.

For example, there may be a rule in MAS_MATCH defined for certain source and target message types that looks like this: {n6 1}:\2\0{n6<!=1>}. When processing this particular rule, the MFL translation function does the following: validate that the source field or variable is a six-digit number; make the first two characters of the target field or variable "20" and then append whatever from the source field or variable at the end. This is actually a rule to translate a date with a two-digit year to the same date with the four-digit year. To make this correct, there is a condition linked to this rule that says "Apply this rule only if the number interpreted from the first two digit of the source field or variable is between 00 and 49." There is, of course, another rule which insert the number "19" in front of the original date if this condition is no satisfied.

The MFL may further include the following pre-loading functions:

LoadMessage();

LoadMessageGroup() and

LoadMatch().

The LoadMessage() specifications may be as follows:

```
int LoadMessage (
    [in] int    Aid,
    [in] int    MsgId)
```

The LoadMessageGroup() specifications may be as follows:

```
int LoadMessageGroup (
    [in] int    Aid)
```

The LoadMatch() specifications may be as follows:

```
int MasMsgFormat (
    [in] int MatchId)
```

5 LoadMessage(), LoadMessageGroup(), and LoadMatch() are used to pre-load message format information from database at application initialization phase. It avoids loading format information when processing message (load-on-demand). Pre-loading technique improves system run-time performance.

LoadMessage(), LoadMessageGroup(), and LoadMatch() are backward compatible. MFL functions can still load format information whenever necessary.

10 The first parameter, Aid, specifies the application Id, and MsgId specifies the message id of the SWIFT message. The MatchId is the ID in the MAS_MATCH table. The return error code can be used to call GetMFL_ERRORS().

In addition, the MFL may include a GetMFL_ERRORS() error functions. The GetMFL_ERRORS() specifications may be as follows:

```
15 long GetMFL_ERRORS (
    [in] long ErrorReference
    [out] char***ErrorDetail)
```

If the parse or format functions return an error value, the number returned is the ErrorReference number. When this number, multiplied by -1, is passed into 20 GetMFL_ERRORS(), MFL looks up the number and returns an array of strings that describes the errors that were encountered in the original call.

Preferably the MFL further includes dynamic link library (DLL) functions such as the following:

```
FreeMFLString();
25 FreeMFLStringArray(); and
FreeMFLIntArray().
```

The FreeMFLString() specification may be as follows:

```
DllImport long FreeMFLString (
    char* str);
```

30 The FreeMFLStringArray() specification may be as follows:

```
DllImport long FreeMFLStringArray (
    char** sarray,
```

```
int    count);
```

The FreeMFLIntArray() specification may be as follows:

```
DllImport long FreeMFLIntArray (
```

```
int*    iarray,
```

```
5      int    count);
```

FreeMFLString() frees a single string allocated by the DLL.

FreeMFLStringArray() (FreeMFLIntArray(), respectively) frees an array of strings (integers) allocated by the DLL (count is the number of elements of the array).

10 The MFL uses regular expressions extensively in meta-data definition, message parsing, formatting and translation. Preferably the regular expressions follow the following rules.

Any literal character is prefixed with escape symbol '\.'

'a' represents any character which can be used in a field, including alphanumeric characters and others, but cannot be any of these five characters: \n - : } \0.

'c' represents alphabetic characters, namely A-Z or a-z.

'n' represents any digit (0-9), space, '-', or ','.

'd' represents any digit, namely, 0-9.

'r' represents new line.

20 Capital letters A, B, C, D, E, F, G, and H are used for standard SWIFT formats on these format options.

<code_table_name> means any value from the table, <code_table_name length> includes the length of the string when the string length is fixed.

('(',')') are used to group items into one item.

25 '['(',')' group one or a number of items into an optional item.

'|' means "or."

Prefixing a number repeats the item by up to as many times.

Prefixing a number then 'x' repeats the item on up to as many lines.

*, instead of a number, represents arbitrary number.

30 '+', instead of a number, represents arbitrary positive number.

Suffixing a number means exact number.

Suffixing 'x' then a number means exact as many lines.

Do not apply number, '*' or '+' on optional sub-expression, 'r,' < > or literal string to express repetition.

The following table provides examples of basic elements.

| Regular Expression | Meaning | Matching String |
|--------------------|--|------------------------------------|
| \F0\1 | literal string "F01" | "F01" |
| \Fd2 | 'F' followed by 2 digits | "F01", "F32" |
| \{d\.:30a\}[r] | a SWIFT header up to 30 chars long with an optional new line | "{1:F01xxxx}" "{2:I300xxxx}\n" |
| ((\A\B) (\C\D)) | "AB" or "CD" | "AB", "CD" |
| (\A\B\C\D) | same as ((\A\B) (\C\D)) | "AB", "CD" |
| \A(\B \C \D)\E | | "ABE", "ACE", "ADE" |
| 5xd | up to 5 lines, each with single digit | "3\n4\n8\n4\n0\n", "0\n2\n", "" |
| *xd | any number of lines with a single digit on each line | "9\n2\n", "" |
| +x(*a) | any positive number of lines with any string on each line | "ABC\n\n4%\n\$\n", "\n" |
| dx3 | 3 lines with single digit on each | "3\n4\n5\n" |
| (*a)x3 | 3 lines | "\n4@\$\nABC\n" |

5

The regular expression uses capital letters A through H to represent standard SWIFT formats as listed in the following table.

| Shortcut | Expansion |
|----------|---|
| A | ([\V1a][\V34a]ra4a2a2 [\V34a]ra4a2a2 a4a2a2)[a3] |
| B | [\V1a][\V34a]r35a [\V34a]r35a 35a |
| C | \V34a |
| D | [\V1a][\V34a]r4x(35a) [\V34a]r4x(35a) 4x(35a) |
| E | a4[r\V\C \V\D][\V34a]ra4a2(2a)[3a] |
| F | a4[r\V\C \V\D][\V34a]r4x(35a) |
| G | <MAS_C_CURRENCY 3>(15n) n8r<MAS_C_CURRENCY 3>(15n) |
| H | <MAS_C_CURRENCY 3>(15n) n8r<MAS_C_CURRENCY 3>(15n) |
| I | a4a2a2[a3] |
| J | 5x(40a) |

10

The MFL applies these shortcut letters only at the beginning of each regular expression format in database. So an RE like this: "C20d," will be translated to "V34a20d" before being used to parse the incoming messages.

MFL format library functions follow these steps to parse and to format messages:

Retrieve all format information for the message from the database.

5 Build a finite automation (simplified as a tree representation) based on the format information.

Use this finite automation to parse and to format the message.

Each regular expression has its value type. The table below provides a list of legal value types. These value flags reflect to the rules somewhat.

| Value Flag | Meaning | Handled or Not |
|------------|-----------------------|----------------|
| A | any character | |
| C | alphabetic characters | |
| N | any digit or ‘,’ | |
| G | any digit | |
| T | table | |
| B | table number | |
| M | match table | |
| P | match table number | |
| L | literal string | |
| I | intermediate node | y |
| O | alternative | y |
| R | new line | |
| D | body of message | |
| S | sub-fields | |
| | | |

10 Examples of regular expression (RE) and their corresponding tree are described below and in Figures 28-31.

Concatenation is denoted by Intermediate node, whereas alternative is denoted by Or node, e.g., RE = “S1 S2 | S3 S4 | S5 | S6 S7 S8” where Sn are single element sub-expressions. A tree built on this expression is depicted in Figure 28. The tree depicted in Figure 28 shows that concatenation takes precedence over alternative, and alternative is right associated. So the preceding RE is interpreted as: (S1 S2) | ((S3 S4) | (S5 | (S6 S7 S8))).

The following sections describe various types of single element RE’s and their corresponding trees.

A Parentheses pair is denoted by Intermediate node, e.g., RE = "S0 (S1 | S2) S3," where Sn are single element sub-expressions. The tree built on this expression is shown in Figure 29.

5 An Option pair is denoted by om_flag on the node, which represents the optional part, e.g., RE = "S0 [S1 | S2] S3," where Sn are single element sub-expressions. The tree built on this is shown in Figure 30. The om_flag on the middle Or node is set to 'O,' which means optional.

Variable assignment on any part of the RE is denoted by var_map_type, var_num, and var_scope on the node representing the assigned part, e.g., RE = "S0 {S1S2 num} S3," where Sn are single element sub-expressions. The tree built on this
10 expression is shown in Figure 31.

The flags on the middle intermediate node (not root I node) is set to following values for different types of variable assignment. If there is no optional part "<!=name_," only var_num is set.

15 { S1S2 [<!= []] num } var_num = num. var_map_type = 'E,' var_scope = '_' ?
1 : 0
{ S1S2 [<!, []] num } var_num = num. var_map_type = 'D,' var_scope = '_' ?
1 : 0
{ S1S2 [X!name []] num } var_num = num, var_map_type = 'T,'
20 tab_map.name = name, var_scope = '_' ? 1 : 0

Any structured message can be represented in the MFL format database.

Currently, most formats are related to SWIFT I or II, or SWIFT messages wrapped by GRN envelope.

MFL format database, which is referred to as meta-data, may include tables.

25 In the embodiment described herein there are 21 tables in the MFL format database. MAS_SYSTEM is for database version information. The other 20 tables are described below.

Information on supported versions:

30 MAS_MSG_VERSIONS
message_version integer,
version_name char(32),
version_date datetime,
effective_date datetime,

| | |
|-----------------|---------------|
| flags | char(16), |
| message_format | varchar(255), |
| message_format1 | varchar(255), |
| comments | text |

5

Version number is assigned by MFL for different versions of SWIFT messages. It can be utilized for other purposes. For instance, different formats, such as CHIPS or FedWire, can be considered as versions in the database. Version date is the official date when it is adopted as standard. The effective date is the date when it is adopted in CrossMar applications. See the next section for a list of some message versions currently in the MFL format database.

10

Each message consists of an envelope and a message body. In the version format for any given message, "<<BODY>>" denotes the body of the message and everything else is the envelope of the message.

15

A list of defined variables

MAS_MSG_VARIABLES

| | |
|-----------------|--------------|
| message_version | integer, |
| var_number | integer, |
| var_name | varchar(32), |
| comments | varchar(255) |

20

This table lists all variables defined in the envelope and body of each message version.

25

A list of supported messages:

MAS_MESSAGES

| | |
|-----------------|--|
| message_version | integer, |
| message_number | integer, |
| message_name | char(32), |
| enforce_order | char(1) value in ('Y,' 'N') default 'Y,' |
| comments | text |

30

35

For SWIFT messages, message numbers are the same as assigned by SWIFT. For example, the message number of SWIFT MT100 is 100. When enforce_order is 'Y,' the fields in a message body have to be in their positions defined in MAS_GEN_FORMAT and MAS_B_GEN_FORMAT (see below.

40

General message format:

MAS_GEN_FORMAT

| | | |
|----|-----------------|--|
| | message_version | integer, |
| 5 | message_number | integer, |
| | position | integer, |
| | id | integer, |
| | name | char(32), |
| | format_flag | char(1), |
| 10 | field_block | char(1) value in ('B,' 'F') default 'F,' |
| | M_O | char(1) value in ('M,' 'O'), |
| | repeat | integer default 1, |
| | comments | text |

15 Position is an index. It is useful for mutually exclusive fields: when two fields share one index, only one can be present. Block means a collection of fields. When field_block field is of value 'B,' the record represents a block. Repeat means that this field can repeat up to as many times (0 means arbitrary times). Field id and format_flag correspond to the tag number in SWIFT specifications.

20

A list of blocks:

MAS_GEN_BLOCK

| | | |
|----|---------------|-----------|
| | block_id | integer, |
| 25 | block_name | char(32), |
| | enforce_order | char(1), |
| | comments | text |

Block number is assigned in sequence. It is preferably unique across board.

30

General block format:

MAS_B_GEN_FORMAT

| | | |
|----|-------------|------------------------------|
| | block_id | integer, |
| 35 | position | integer, |
| | id | integer, |
| | name | varchar(32), |
| | format_flag | char(1), |
| | field_block | char(1), |
| 40 | M_O | char(1) value in ('M,' 'O'), |
| | repeat | integer default 1, |
| | comments | varchar(255) |

Same as MAS_GEN_FORMAT except that everything is for blocks.

Application specific information:

MAS_APPLICATION

| | | |
|----|-----------------|-----------|
| 5 | app_id | int, |
| | message_version | int, |
| | sample_flag | char(1), |
| | app_date | datetime, |
| | effective_date | datetime, |
| 10 | comments | text |

This is a list of mapping from application id's to version numbers.

MAS_MESSAGE_TITLE

| | | |
|----|-----------------|-------------|
| 15 | message_number | integer(4), |
| | app_id | integer(3), |
| | message_version | integer(3), |
| | caption | char(64), |
| | comments | text |

MAS_BLOCK_TITLE

| | | |
|----|----------|-------------|
| 20 | block_id | integer(4), |
| | app_id | integer(3), |
| | caption | char(64), |
| 25 | comments | text |

Field formats for all fields not in any block:

MAS_APP_FIELD

| | | |
|----|----------------|--|
| 30 | message_number | integer(4), |
| | field_number | integer(3), |
| | app_id | integer(3), |
| | format_flag | char(1), |
| | position | integer, |
| 35 | subfield_exist | char(1) value in ('Y,' 'N'), |
| | field_format | char(128), |
| | M_O | char(1) value in ('M,' 'O'), |
| | caption | char(64), |
| | read_only | char(1) value in ('R,' 'A') default 'A,' |
| 40 | app_flags | char(3), |
| | indx_field | char(1), |
| | comments | text |

Field formats for all block fields:

MAS_B_APP_FIELD

| | | |
|----|--------------|--|
| | block_number | integer(4), |
| | field_number | integer(3), |
| 5 | app_id | integer(3), |
| | format_flag | char(1), |
| | position | integer, |
| | field_format | char(128), |
| | M_O | char(1) value in ('M,' 'O'), |
| 10 | caption | char(64), |
| | code_table | char(32), |
| | read_only | char(1) value in ('R,' 'A') default 'A,' |
| | disp_pref | char(1) |

15 Formats for subfields in a non-block fields:

MAS_APP_SUBFIELD

| | | |
|----|----------------|--|
| | message_number | integer(4), |
| | field_number | integer(3), |
| 20 | app_id | integer(3), |
| | format_flag | char(1), |
| | subfield_tag | char(16), |
| | field_format | char(128), |
| | M_O | char(1) value in ('M,' 'O'), |
| 25 | caption | char(64), |
| | read_only | char(1) value in ('R,' 'A') default 'A,' |
| | app_flags | char(3), |
| | comments | text |

30 Formats for subfields in a block field:

MAS_B_APP_SUBFIELD

| | | |
|----|--------------|---------------|
| | block_number | int, |
| | field_number | int, |
| 35 | app_id | int, |
| | format_flag | char(1), |
| | subfield_tag | varchar(16), |
| | field_format | varchar(128), |
| | M_O | char(1), |
| 40 | caption | varchar(64), |
| | read_only | char(1), |
| | app_flags | char(3), |
| | comments | text |

- 102 -

A list of all codes used in MFL format database:

MAS_CODE_TABLES

5 table_name varchar(32),
 code varchar(16),
 description varchar(64),
 comments varchar(128)

A list of combined or base rule clause identifiers for cross-field validation rules and
 10 translation related rules:

MAS_RULE

15 app_id int not null,
 message_number int not null,
 condition1 int not null,
 condition2 int not null,
 err_code varchar(8) not null,
 comments varchar(255) null

20 A list of binary combinations of base rule clauses:

MAS_RULE_COMBINE

25 clause_ref int not null,
 operator char(1) not null,
 component1 int not null,
 component2 int not null

A list of base rule clauses:

MAS_RULE_CLAUSE

30 clause_ref int not null,
 block_id int not null,
 field_block char(1) not null,
 field_number int not null,
 35 format_flag char(1) null,
 position int not null,
 operator char(1) not null,
 reg_exp varchar(128) null,
 40 comments varchar(255) null

A list of translation and matching identifiers and their applications:

MAS_MATCH_TITLE

| | | | |
|----|----------------|--------------|-----------|
| | match_id | int | not null, |
| | match_name | varchar(32) | not null, |
| 5 | match_type | char(1) | not null, |
| | app_id1 | int | not null, |
| | message_block1 | char(1) | not null, |
| | message_id1 | int | not null, |
| | app_id2 | int | not null, |
| 10 | message_block2 | char(1) | not null, |
| | message_id2 | int | not null, |
| | condition | int | not null, |
| | threshold | float | not null, |
| 15 | comments | varchar(255) | null |

A list of all the translation and matching rules:

MAS_MATCH

| | | | |
|----|----------------|--------------|-----------|
| | match_id | int | not null, |
| 20 | field_var1 | char(1) | not null, |
| | field_number1 | int | not null, |
| | format_flag1 | char(1) | null, |
| | position1 | int | not null, |
| | condition1 | int | not null, |
| 25 | match_operator | char(1) | not null, |
| | match_rule | varchar(255) | null, |
| | field_var2 | char(1) | not null, |
| | field_number2 | int | not null, |
| | format_flag2 | char(1) | null, |
| 30 | position2 | int | not null, |
| | condition2 | int | not null, |
| | weight | float | not null, |
| | comments | varchar(255) | null |

35 A list of all mapping tables used in translation and matching:

MAS_MATCH_TABLES

| | | | |
|----|------------|--------------|-----------|
| | table_name | varchar(32) | not null, |
| | from_code | varchar(32) | not null, |
| 40 | to_code | varchar(32) | not null, |
| | comments | varchar(128) | null |

The foregoing tables are utilized by the message format library in the translation, parsing, formatting, reconciliation and validating of messages.

As described in detail above, the message format library may be incorporated with the architecture of the system of the present invention in different ways. The message format library may be linked as a local function call to application programs utilizing the system. Alternatively, the message format library may be linked as a remote function call be present on the message agent server. It should also be understood that the message format library of the present invention is a distinct and separate invention which is capable of operating independently of the message agent server described herein.

Embodiments of the present invention have been described above to illustrate the features and advantages of the present invention. It will be appreciated that these examples are merely illustrative of the invention. Variations and modifications will be apparent to those skilled in the art.

Claims

1 1. A method for transferring electronic messages in a network, the
2 network including a message agent server, a first terminal for communicating
3 electronic messages in a first format, and a secure network, the secure network further
4 connectable to at least a second terminal for communicating electronic messages in a
5 secure network format, comprising:

6 generating an electronic message at the first terminal in the first format;

7 transmitting the electronic message from the first terminal to the message
8 agent server;

9 the message agent server automatically translating the electronic message from
10 the first format to the secure network format; and

11 transmitting the translated electronic message from the message agent server
12 to the second terminal.

1 2. The method of claim 1 further comprising:

2 generating a second electronic message at the second terminal in the secure
3 network format;

4 transmitting the electronic message from the second terminal via the secure
5 network to the message agent server;

6 the message agent server automatically translating the electronic message from
7 the secure network format to the first format; and

8 transmitting the translated second message from the message agent server to
9 the first terminal.

1 3. The method of claim 1 wherein the first terminal comprises a
2 computer.

1 4. The method of claim 1 wherein the server comprises a computer.

1 5. The method of claim 1 wherein the second terminal comprises a
2 computer.

1 6. The method of claim 3 wherein the computer is a personal computer.

1 7. The method of claim 1 wherein generating the electronic message at
2 the first terminal in the first format further comprises using a software application
3 such that the first format includes data formatted using the software application.

1 8. The method of claim 1 wherein the secure network comprises a
2 financial network.

1 9. The method of claim 1 wherein the secure network format is a SWIFT
2 compatible format.

1 10. The method of claim 1 wherein the first message includes a plurality of
2 first message elements and the translated electronic message includes a plurality of
3 secure network message elements, further comprising the message agent server
4 automatically parsing the message, such that the plurality of message elements
5 correspond to the plurality of secure network message elements.

1 11. The method of claim 1 further comprising validating the electronic
2 message.

1 12. The method of claim 1 further comprising validating the translated
2 electronic message.

1 13. The method of claim 1 wherein the second terminal is disconnectable
2 from the secure network and wherein transmitting the translated electronic message
3 further comprises connecting the second terminal to the secure network,
4 the server automatically storing the translated message;
5 the server automatically transmitting an alert message to the second terminal;
6 and
7 the second terminal retrieving the translated message.

1 14. A method for transferring electronic messages between a first terminal, a
2 message agent server and a first network comprising;
3 generating an electronic message addressed to the first network on the first
4 terminal;
5 sending the message from the first terminal to the message agent server;
6 interpreting the message so as to determine the address of the electronic
7 message;
8 routing the message to the first network from the message agent server.

1 15. The method of transferring electronic messages of claim 14 further
2 comprising:
3 queuing the electronic message in the message agent server before it is routed
4 to the first network.

1 16. The method of transferring electronic messages of claim 15 wherein the
2 step of routing is delayed until the first network comes on-line such that the message
3 will remain queued until the first network comes on-line.

1 17. A system for transferring electronic messages in a network, the
2 network including a message agent server, a first terminal for communicating
3 electronic messages in a first format, and a secure network, the secure network further
4 connectable to at least a second terminal for communicating electronic messages in a
5 secure network format, comprising:
6 means for generating an electronic message at the first terminal in the first
7 format;
8 means for transmitting the electronic message from the first terminal to the
9 message agent server;
10 means for automatically translating the electronic message from the first
11 format to the secure network format; and
12 means for transmitting the translated electronic message from the message
13 agent server to the second terminal.

1

1 18. A system for transferring messages comprising:
2 a message agent server comprised of;
3 at least one queue;
4 a processor which is used to at least determine the destination
5 address of one of the messages; and
6 a plurality of communications links used to carry the messages;
7 a first terminal;
8 a first network coupling the first terminal to a first subplurality of
9 communications links;
10 a second network coupled to a second subplurality of communications
11 links.

1 19. The system of claim 18 where the first terminal holds one of a plurality of
2 application programs.

1 20. The system of claim 19 where one of the plurality of application programs
2 is a securities exchange program.

1 21. The system of claim 18 where the second network is a world-wide
2 network used to transfer financial messages.

1 22. The system of claim 18 where the first terminal is a server.

1 23. The system of claim 18 where the queue is used to store and transfer
2 messages in a particular order destined for the first terminal.

1 24. The system of claim 18 where the queue is used to store and transfer
2 messages in a particular order destined for the second network.

1 25. The system of claim 18 where the message agent server utilizes an MQ
2 application which is used to control the message transfer to and from the first
3 terminal.

1 26. A message agent server used for transferring messages between a first
2 terminal and a network comprising;
3 communications links coupling the message agent server to the first terminal
4 and the network;
5 a processor;
6 a buffer for temporarily holding a message received via the communications
7 links; wherein the processor operates under the control of an interpreting procedure
8 so the processor determines the destination address of the message while the message
9 is in the buffer;
10 a first queue associated with the first terminal;
11 a second queue associated with the network;
12 where the message is transferred from the buffer to the appropriate queue
13 based on the determination made by the processor under the control of the interpreting
14 procedure.

1 27. A system comprising a message agent server coupled to a first network,
2 the network including a first terminal, and coupled to a second network where the
3 second network includes at least one terminal where the system is further comprised
4 of;

5 communications links coupling the message agent server to the first network
6 and coupling the message agent server to the second network;

7 where the message agent server is further comprised of

8 a processor;

9 message receiving means for receiving messages transmitted to the
10 message agent server from the first and second networks via the communications
11 links;

12 means for interpreting the received messages which includes reading
13 the address of the destination of the message;

14 queuing means for storing the received messages and for forwarding
15 the messages to the communications links when told to do so;
16 routing means for routing the messages to the appropriate destination
17 based upon the determination of the means for interpreting.

1 28. The system of claim 27 wherein the means for interpreting the message
2 includes means for validating the message format.

1 29. The system of claim 28 wherein the means for interpreting the message
2 further comprise means for translating the message from a message format to a second
3 message format.

1 30. The system of claim 29 wherein the means for translating comprise a
2 message format library.

1 31. The system of claim 27 wherein the queuing means comprise a database.

1 32. The system of claim 31 wherein the queuing means further comprise at
2 least one of the message archive, retrieval, and re-transmission capabilities.

1 33. The system of claim 27 further comprising means for monitoring message
2 traffic and message agent server performance by an administrator.

1 34. A method for transferring electronic messages among a plurality of
2 participants wherein the participants include a first terminal and a first network and
3 the method uses a message agent server where the method comprises:

4 preparing an electronic message on the first terminal addressed to the first
5 network;

6 sending the message to the message agent server;

7 interpreting the message to determine at least the message addressee;

8 storing the message in a message queue assigned to the message addressee;

9 triggering the first network to connect to the message agent server;

10 routing the message to the remote network after the remote network connects
11 to the message agent server.

1 35. The method of claim 34 wherein interpreting the message further
2 comprises validating the format of the message.

1 36. The method of claim 34 further comprising monitoring message traffic.

1 37. A message format library comprising:
2 means for parsing messages;
3 means for translating messages; and
4 means for validating message formats.

1 38. The message format library of claim 37 wherein the means for validating
2 uses the means for parsing such that the message is divided into its components by the
3 means for parsing so the means for validating reads the components and compares
4 them to a database of allowed components.

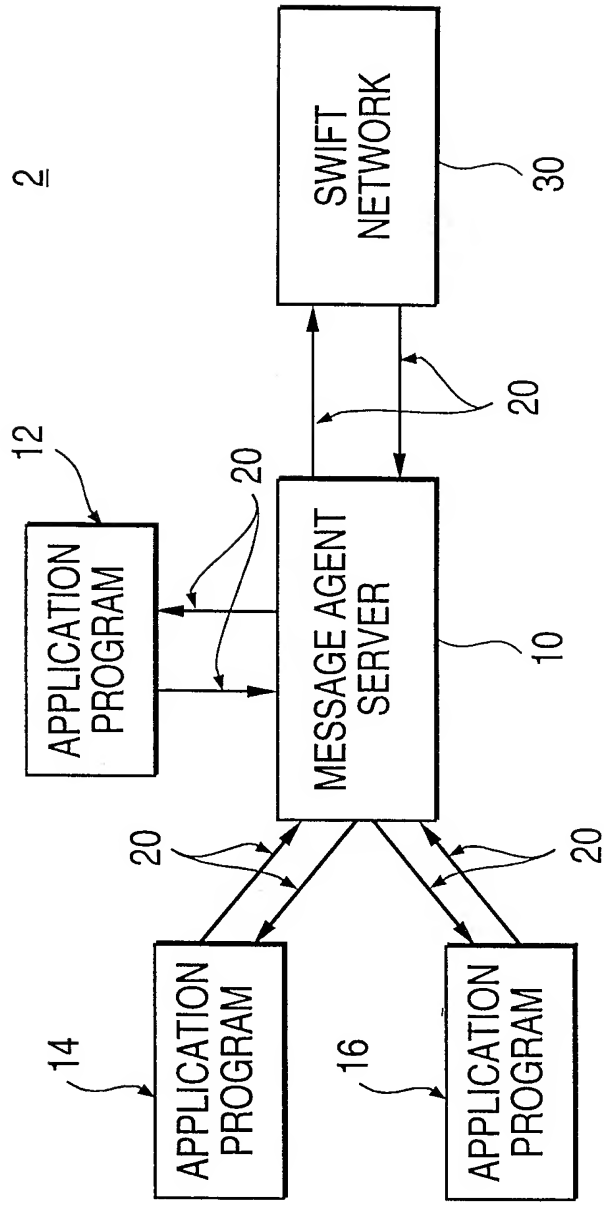
1 39. The message format library of claim 37 wherein the means for translating
2 uses the means for parsing such that the message is divided into its components by the
3 means for parsing so the means for translating may translate the components into a
4 different format based upon a database of acceptable different format components.

1 40. The message format library of claim 37 wherein a destination address
2 within a message is derived by the means for parsing.

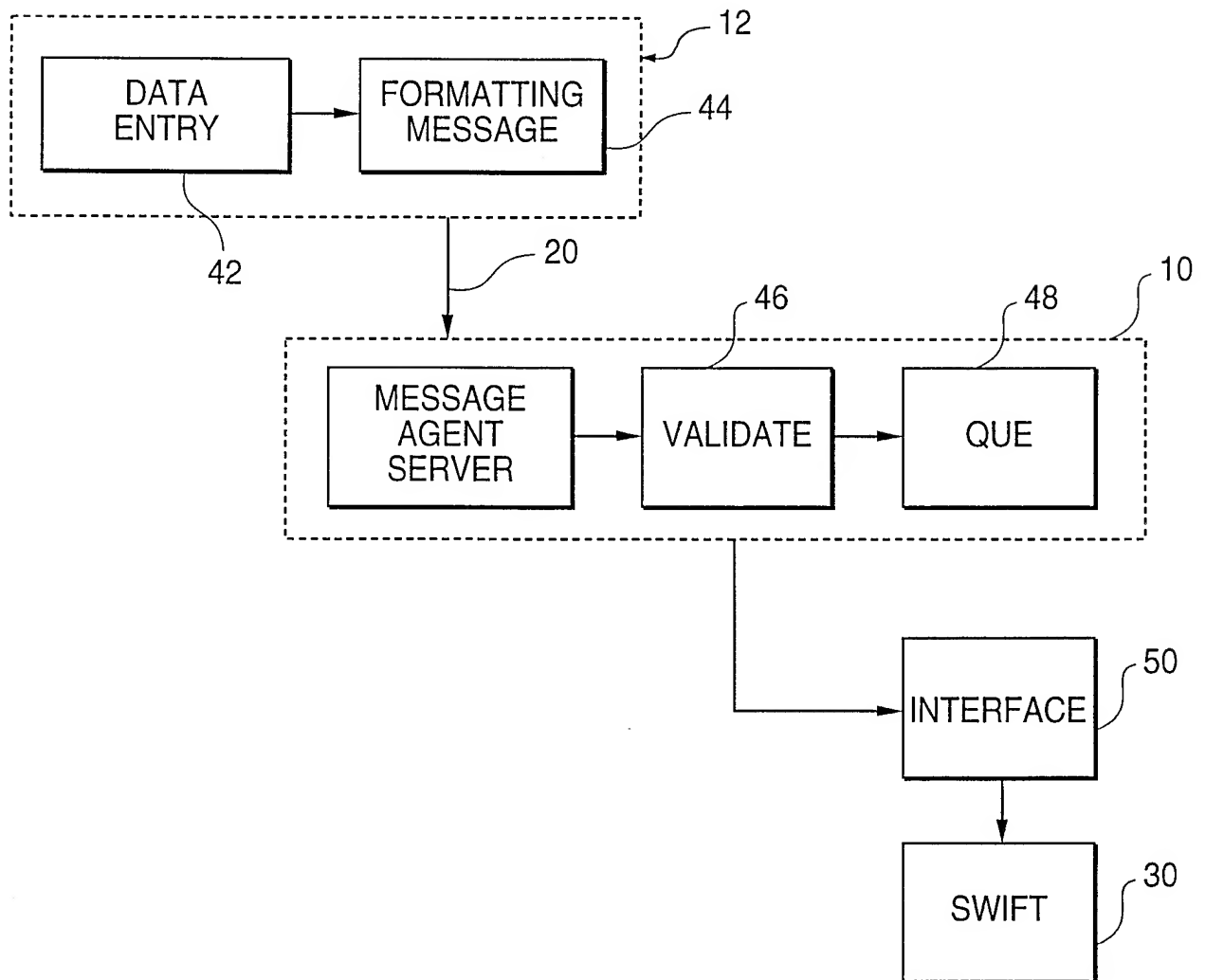
1 41. The message format library of claim 37 wherein a source address within a
2 message is derived by the means for parsing.

1 42. The message format library of claim 37 wherein the means for translating
2 messages translates messages in a first format into a second format and translates
3 messages in the second format into the first format.

FIG. 1



2/28

FIG. 2

3/28

FIG. 3

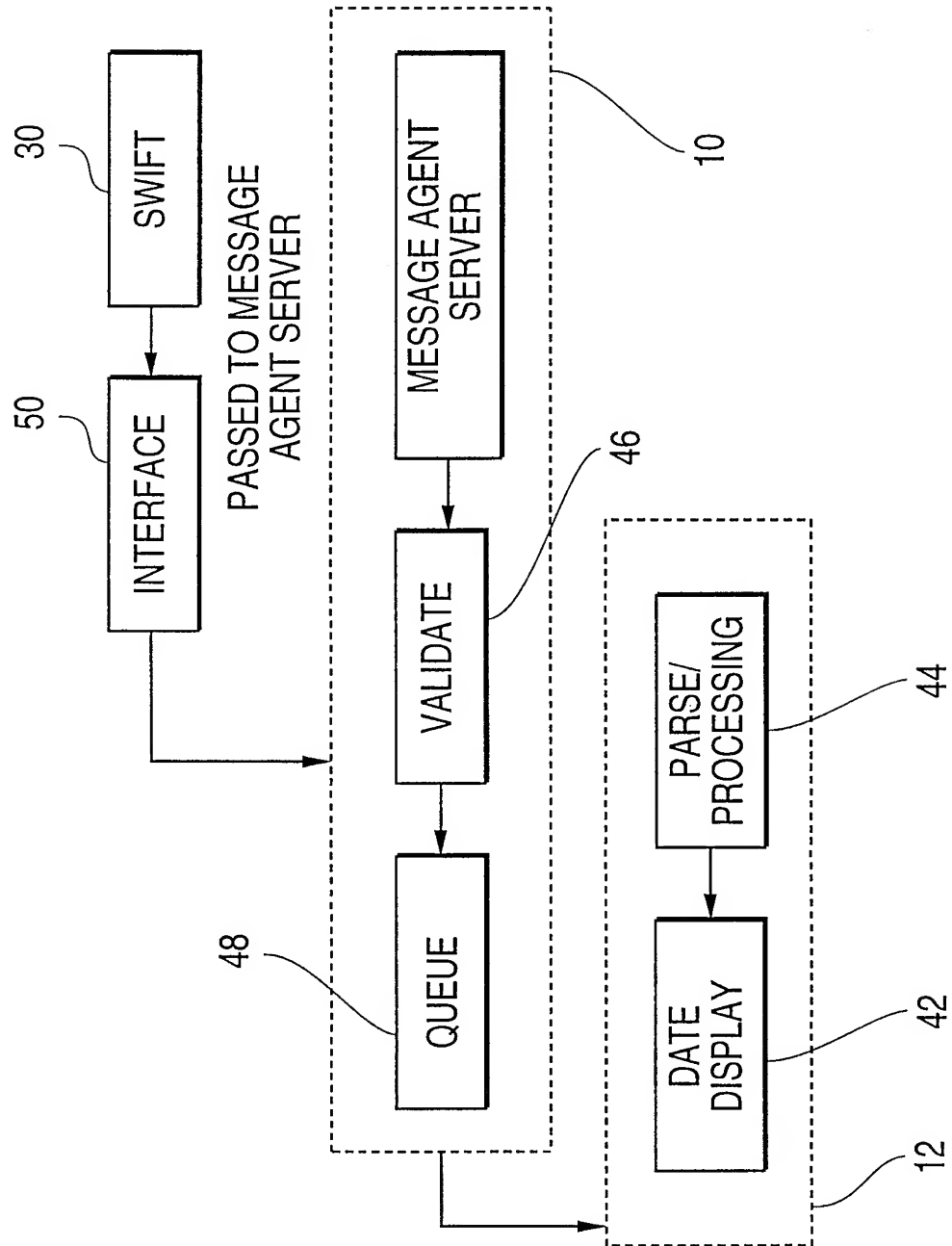
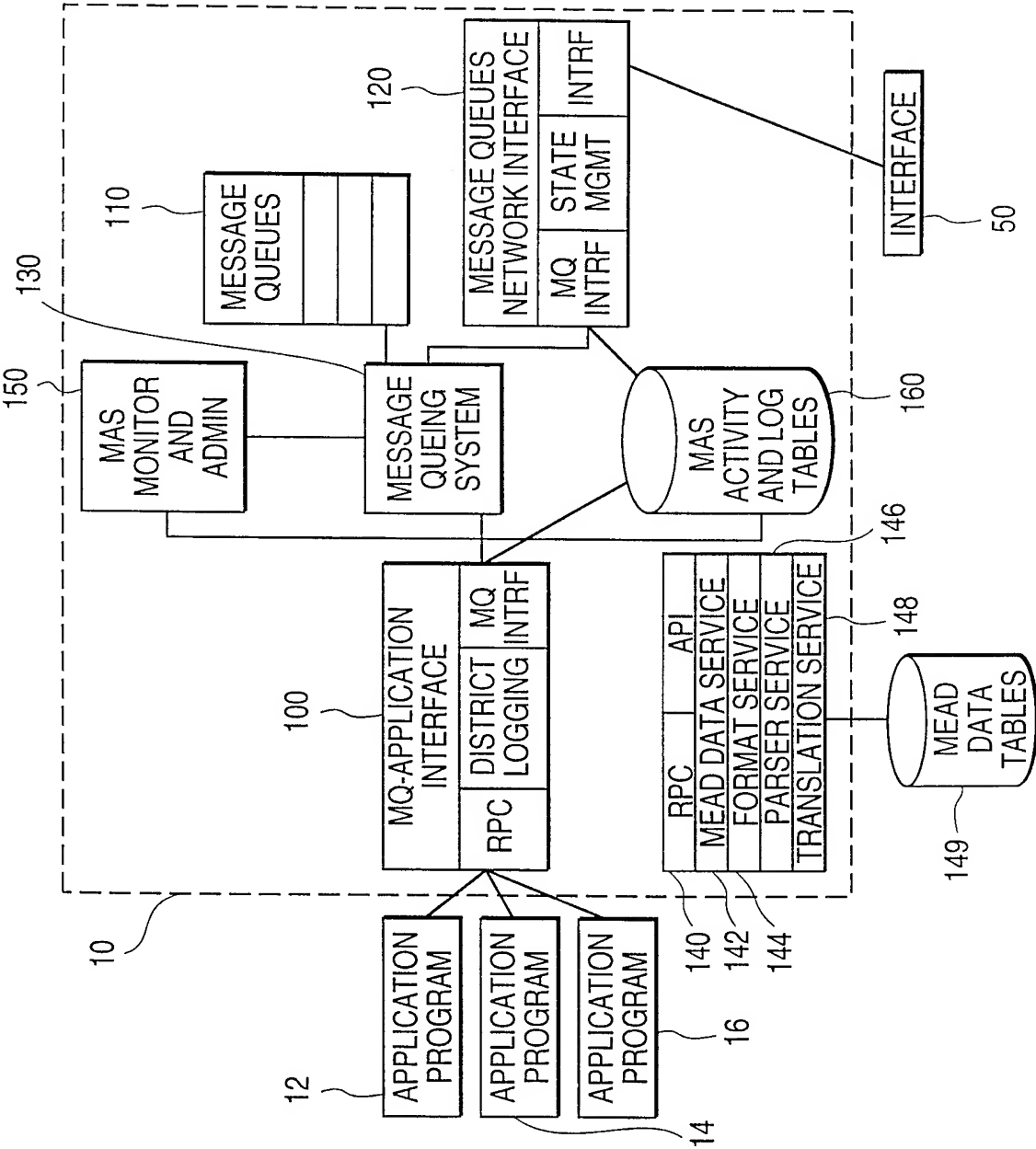
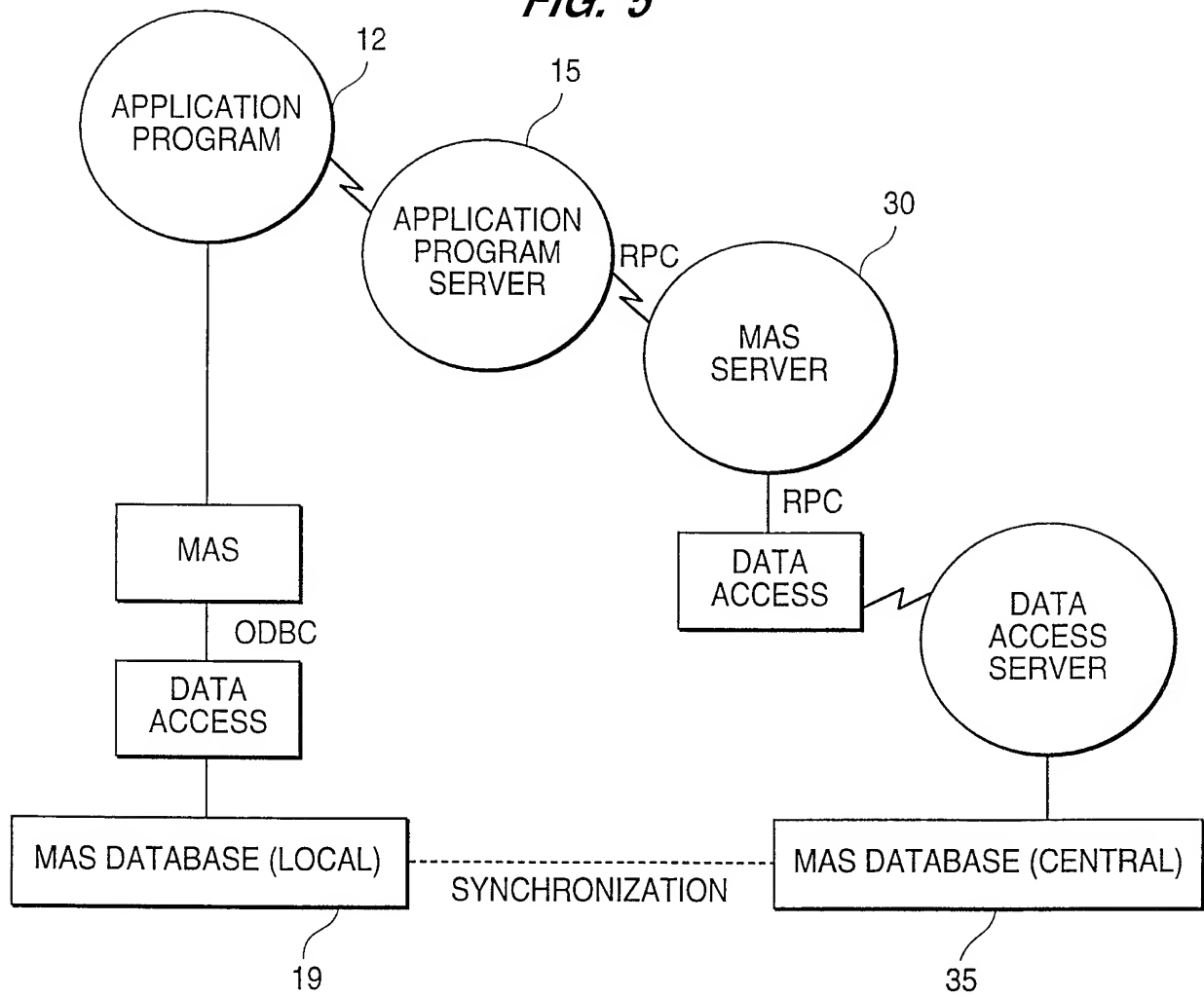


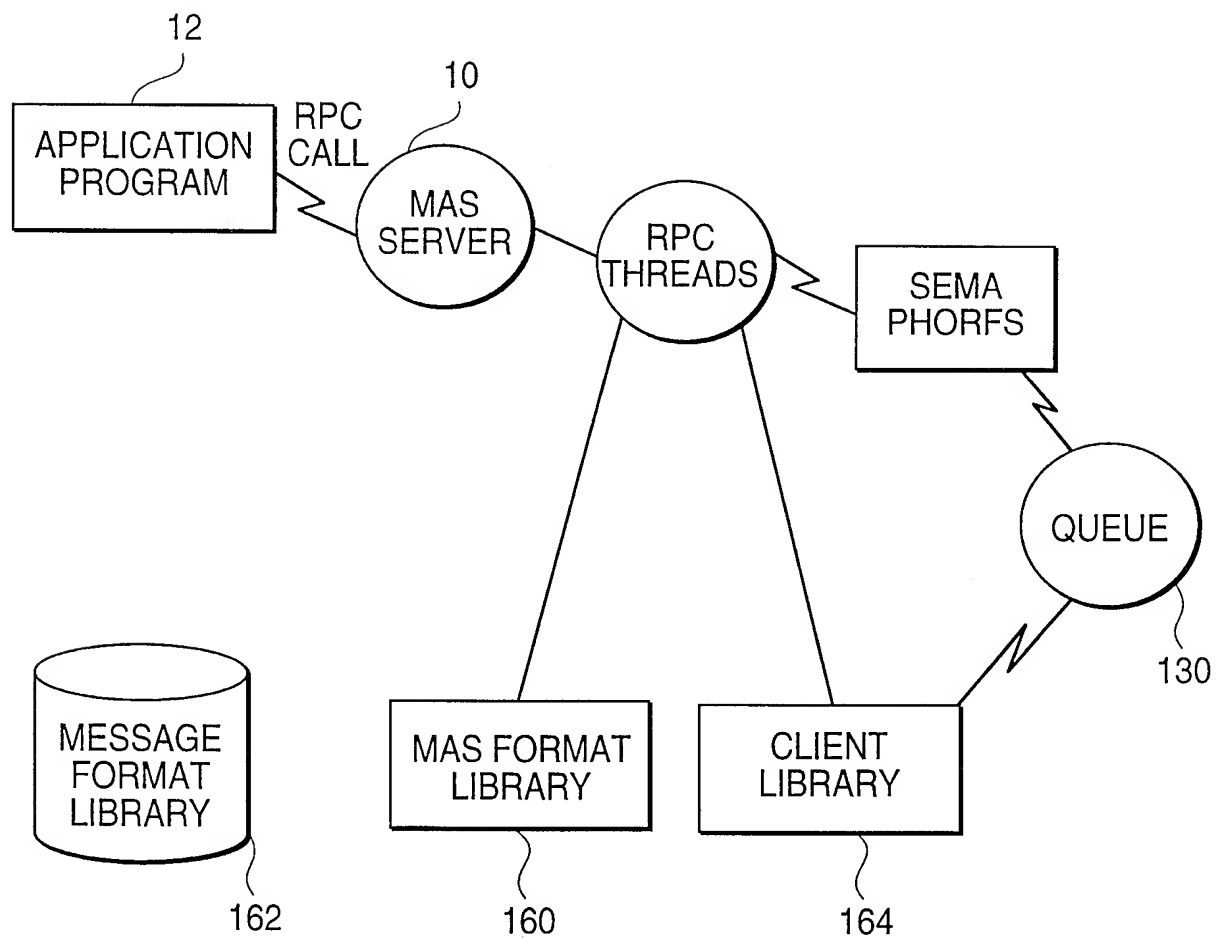
FIG. 4



5/28

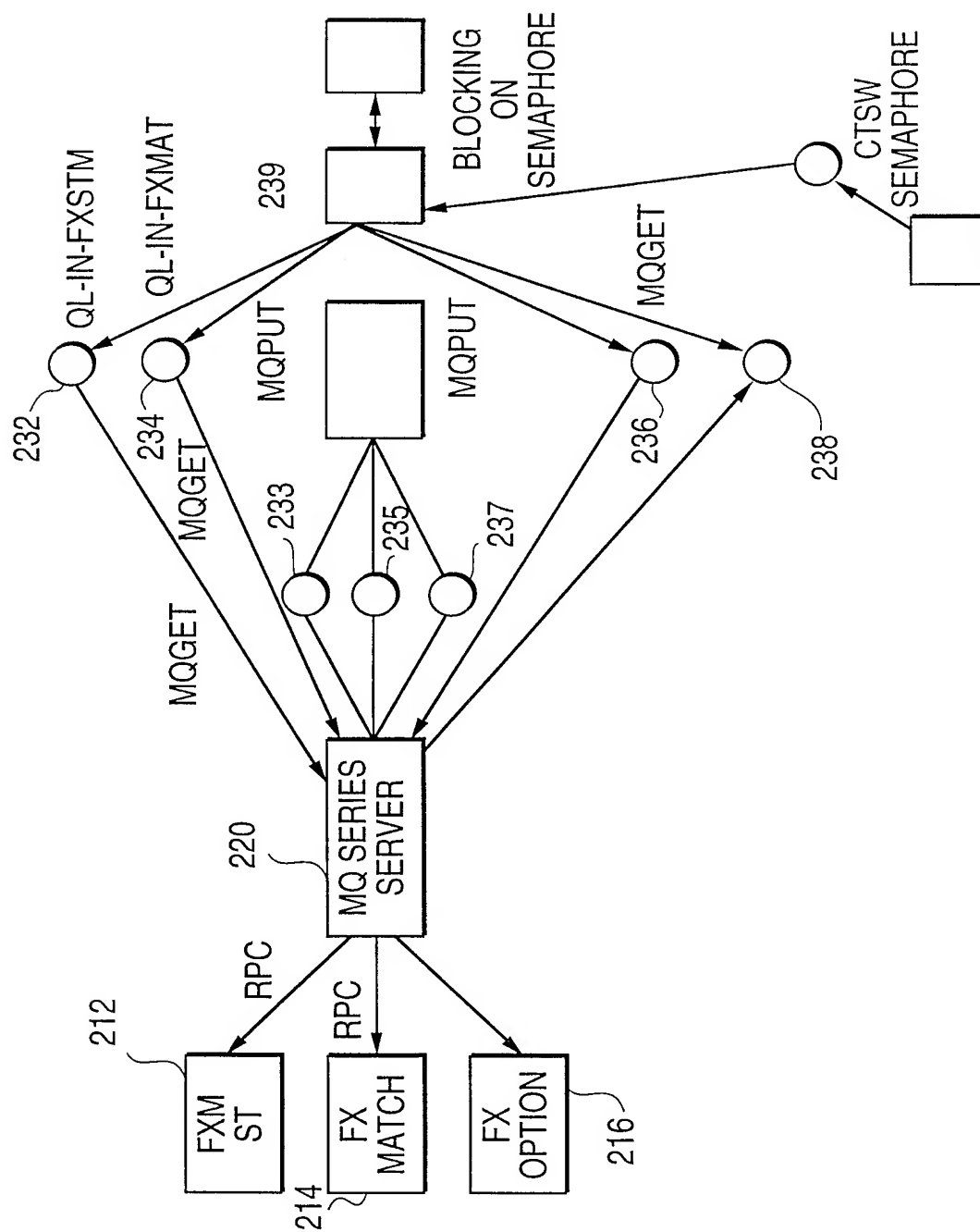
FIG. 5

6/28

FIG. 6

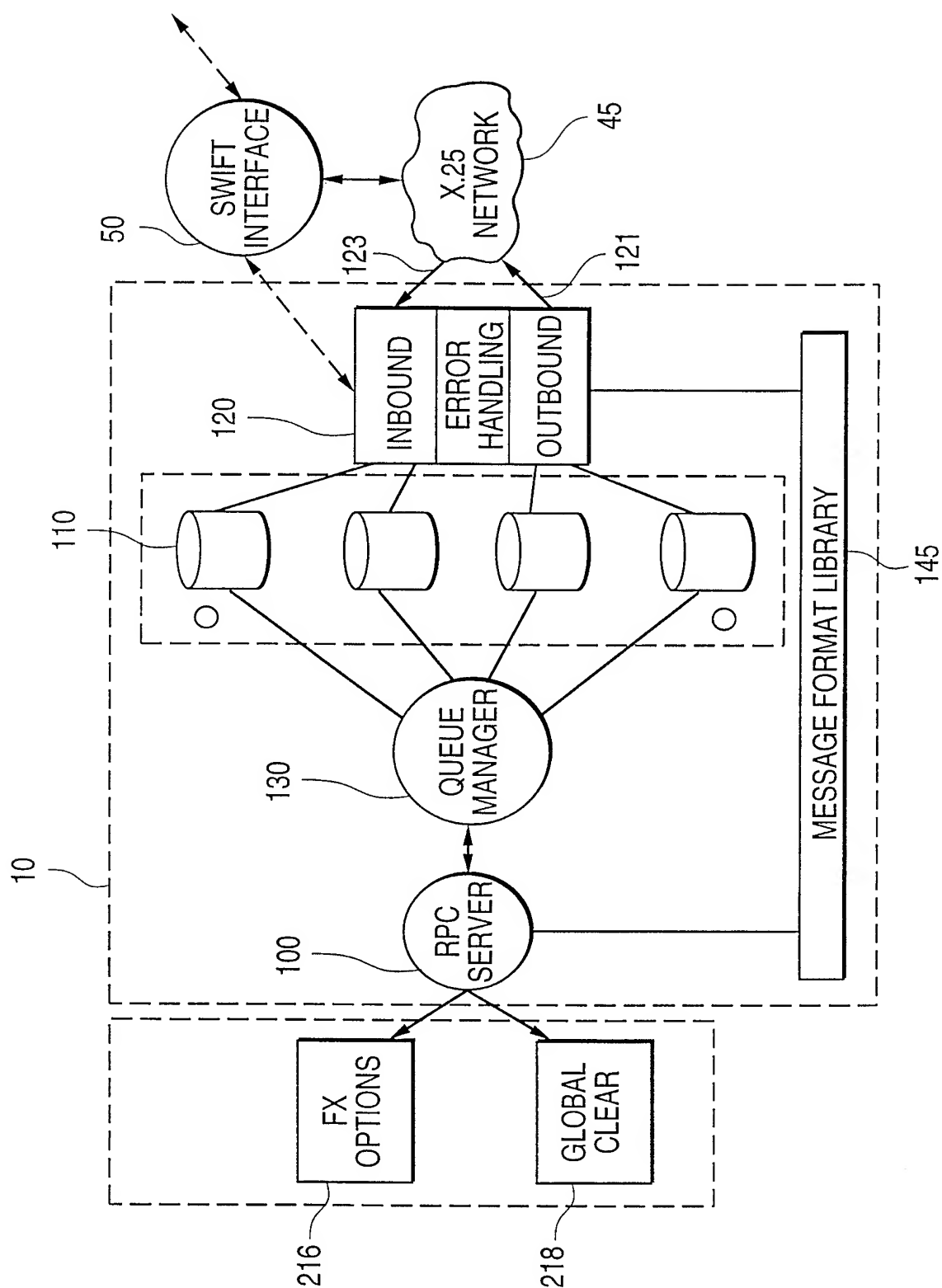
7/28

FIG. 7



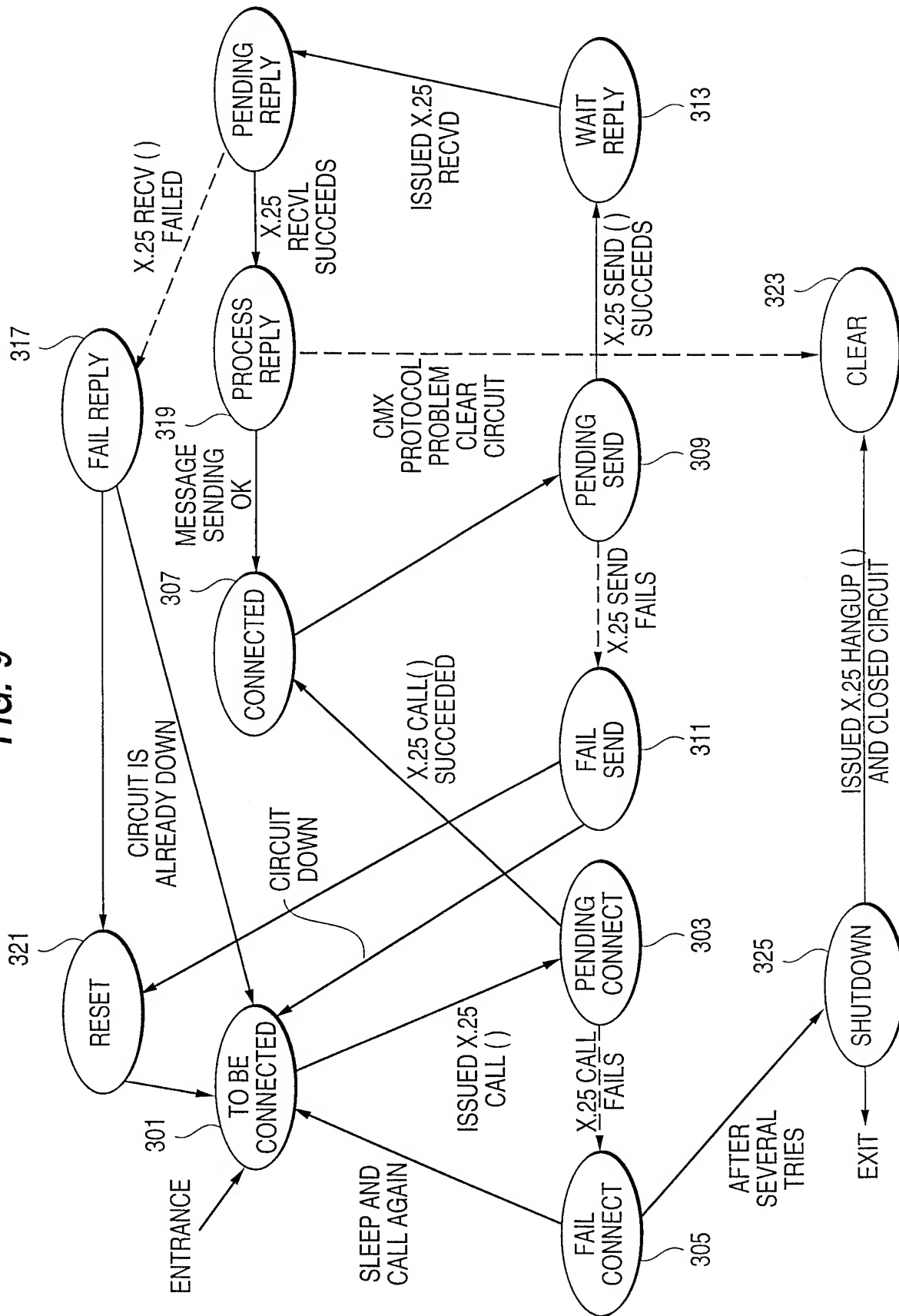
8/28

FIG. 8



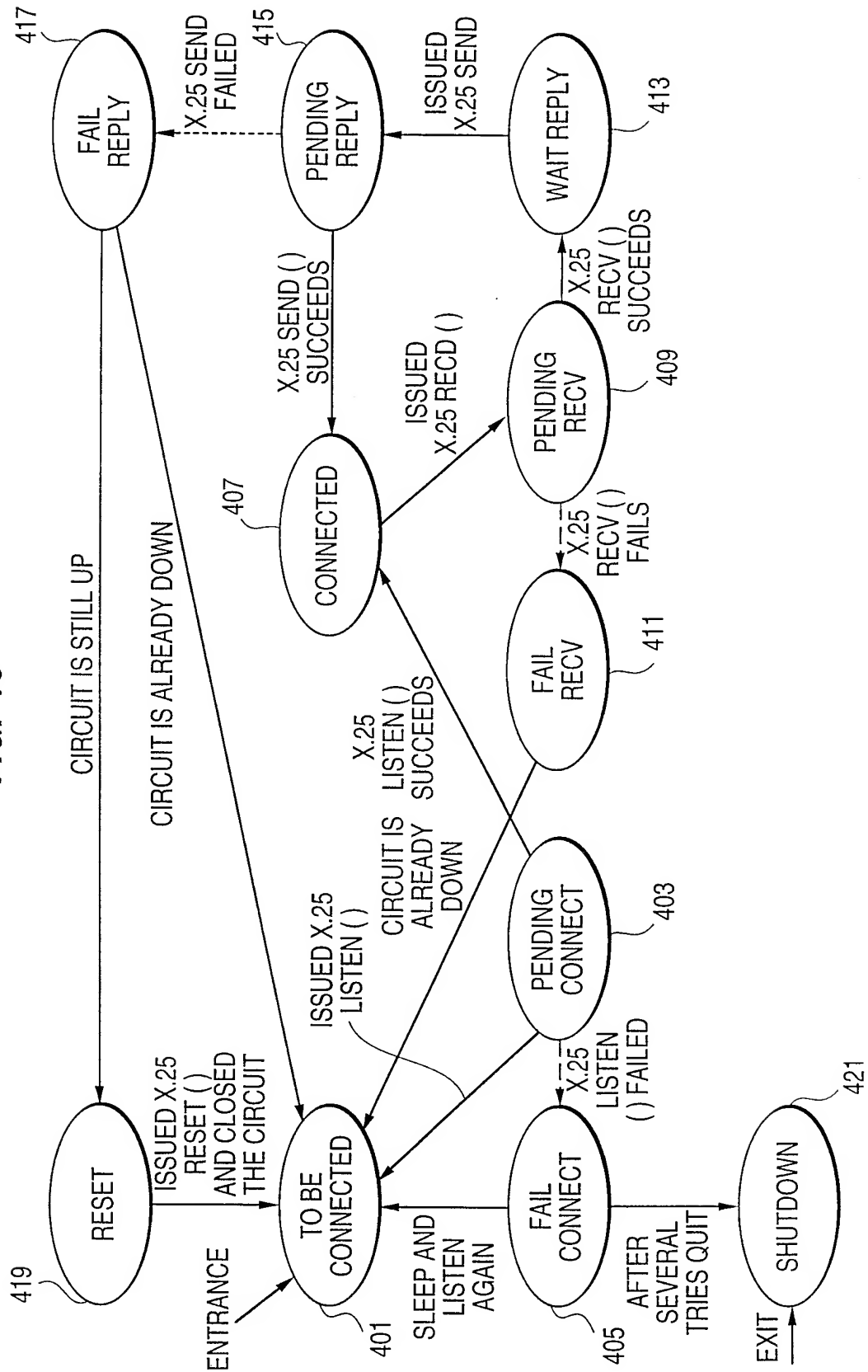
9/28

FIG. 9

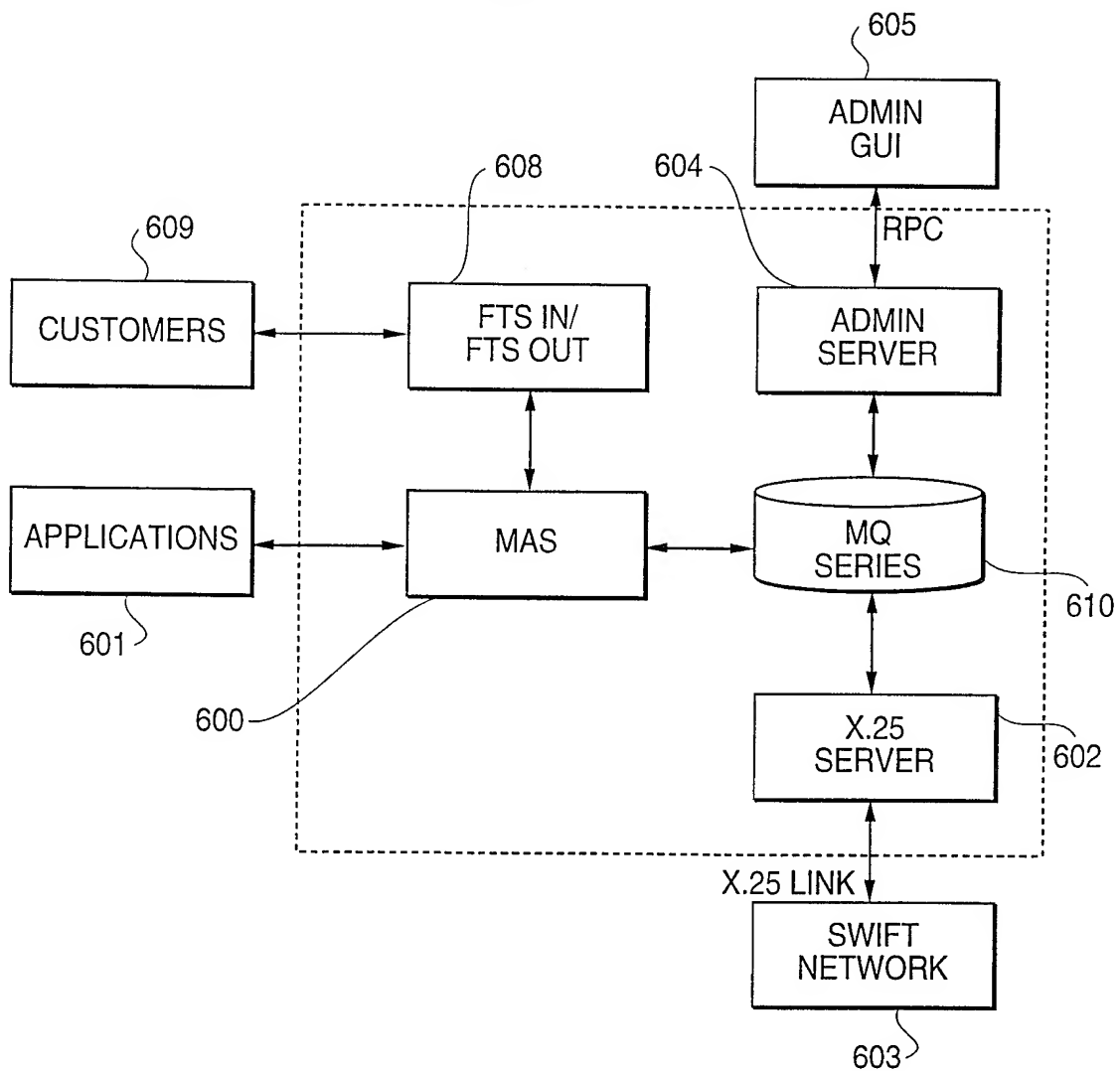


10/28

FIG. 10



11/28

FIG. 11

12/28

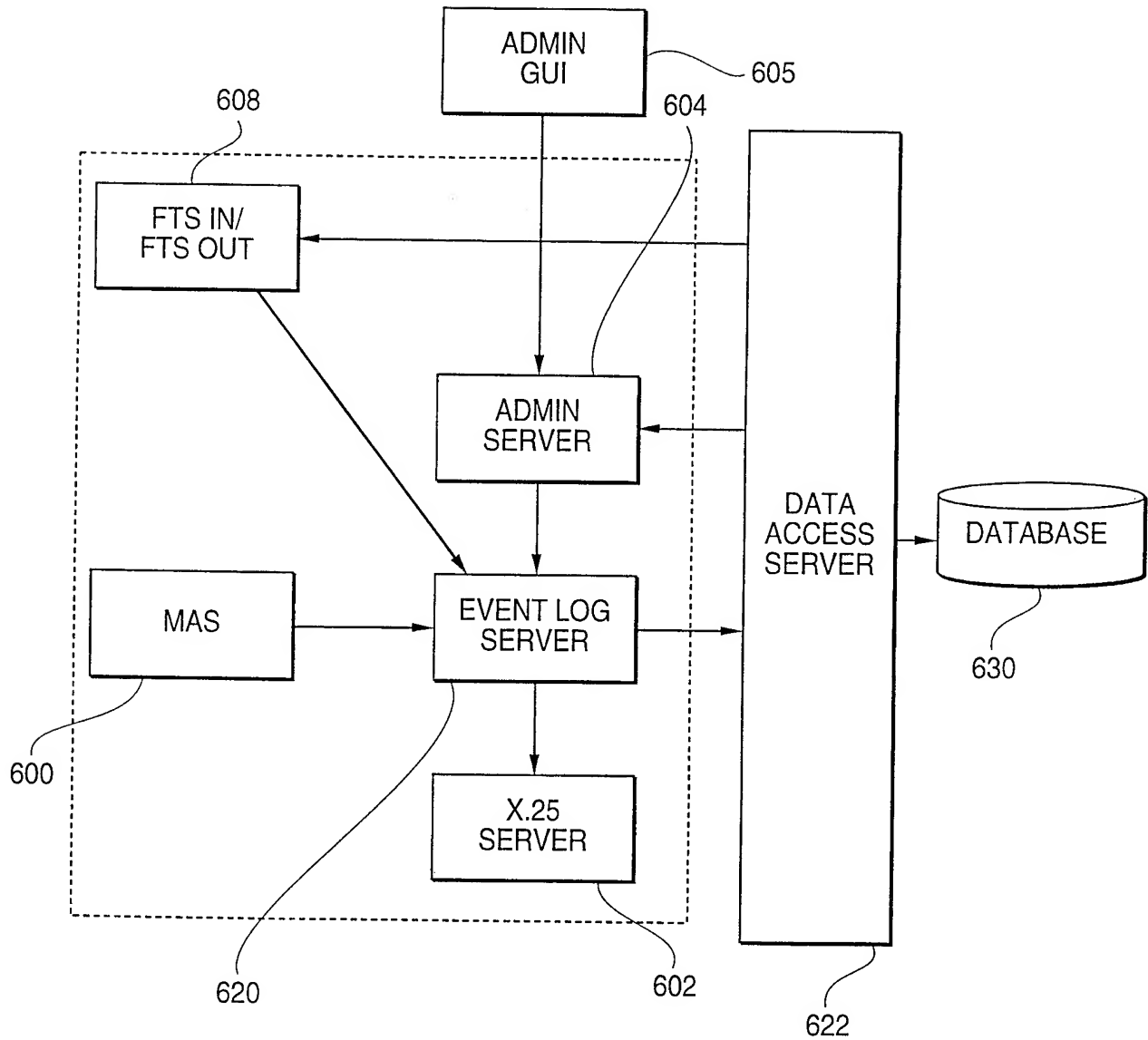
FIG. 12

FIG. 13

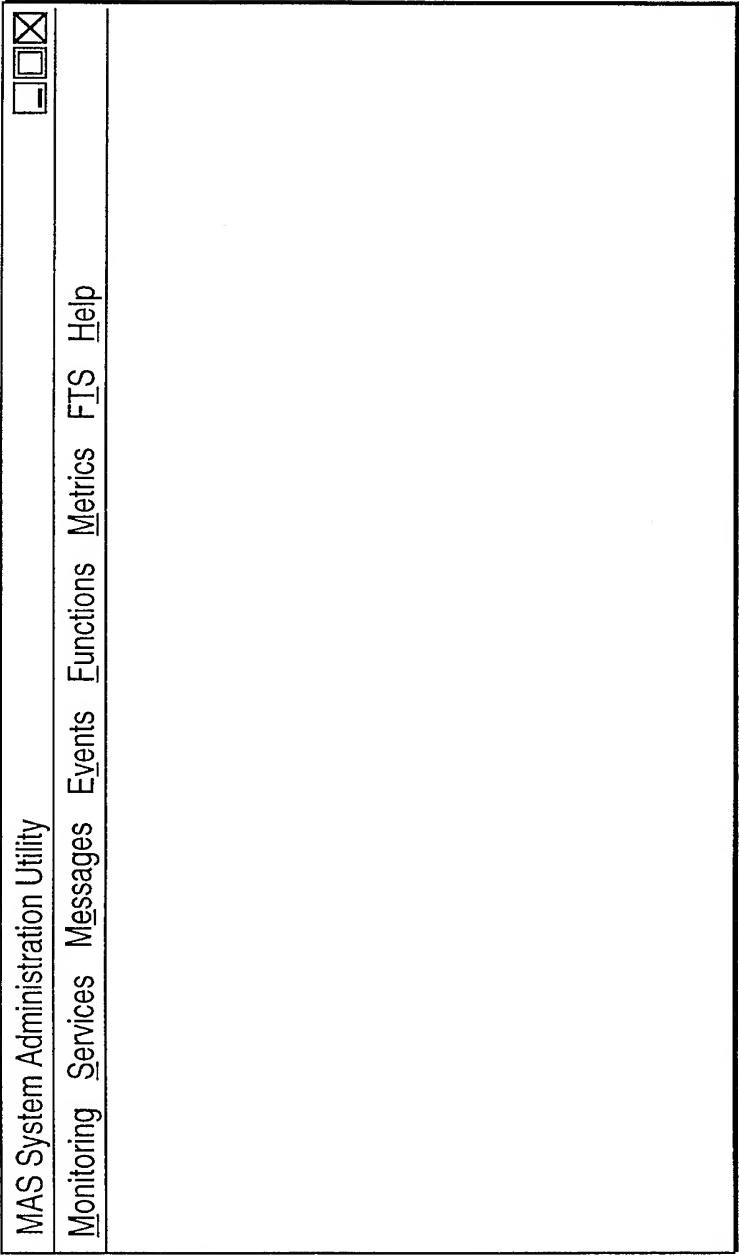


FIG. 14

MAS Monitor

Time of Last Update 17:26:58

☐ Enable audible alarm

Process Status

| Name | Process Name | Status |
|------|--------------|--------|
| VGC | MASDBSvt | UP |
| VGC | MASEvLog | UP |
| VGC | MAS | UP |
| VGC | MASXGWR | DOWN |
| VGC | MASXGWS | DOWN |

Number of Messages on Queue

| Queue Name | Current Depth |
|---------------|---------------|
| QL_IN_ADMIN | 0 |
| QL_IN_FXDPT | 0 |
| QL_IN_FXLNK | 0 |
| QL_IN_FTS | 0 |
| QL_IN_FXMAT | 0 |
| QL_IN_RECON | 0 |
| QL_ERROR_MAS | 1 |
| QL_OUT_CTSW | 1 |
| QL_OUT_STATUS | 3 |

Sequence Numbers/Daily Volume

7 Day Volume Summary

| Station | Seq Number | 04/27 | 04/26 | 04/25 | 04/24 | 04/23 | 04/22 | 04/21 |
|---------|------------|-------|-------|-------|-------|-------|-------|-------|
| GWR | 3828 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| GWS | 0132 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| GXR | 2703 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

15/28

FIG. 15

Error Display

</

FIG. 16

| MAS Event Detail | | | |
|------------------|---------------------|--------------|-------------------|
| Entry Time | Apr 23 1998 3:35:09 | Service Name | MASXGWS |
| Severity | 8 | Event 0 | X25_LowSeq |
| Host Name | CTDVGC | Routine | CMX_VC_process_pa |
| User Name | SYSTEM | Event | 8110 |

ACKNOWLEDGE EVENT CURRENTLY BEING READ

FIG. 17

MAS Services

Start

Stop

Cancel

Hostname:

CTDVGC

Service:

AT

FIG. 18

QL_OUT_CTSW

Queue Name

QL_OUT_CTSW

NYSTS

NYEAACB

TEST NR

(1:F01CITIUS33ARN15017165628)

(2:03001438980417BIMEMXMMAXX196135247598041715

(3:

(108:FXDTST3540900801))

(4:

:15A120:245760121.NEWT122A:NEWT122C:BIMEMM8502

(5:)

OK

Cancel

Browse

Move

Delete

Message ID

FXDTST3540900801

Message insertion

04/24/1998

Message insertion time

14:11:18

FIG. 19

✕

Browse/Move Queued Messages

Queue Name

QL_OUT_MAS ▾

WARNING| POSSIBLE| DUPLICATE|

-0424950 NYX
— TEXT ADDED BY USA, NEW YORK
24-APR 15:51:30 OUTPUT SUPPRESSED
24-APR 15:51:30 SENDER'S REFERENCE "NONE"
24-APR 15:51:30 INPUT SEQ 9381 ON CSI ON FHL003
24-APR 15:51:30 RETURNING FAILED MESSAGE, REASON:
1: SENDER'S SWIFT LT NOT CONFIGURED
-04241950 NYX
— Test added by MAS
04/24/98 15:59:08 Message GWR3831 rejected by MAS due

▴

▾

OK

Cancel

Browse

Move

Delete

Message ID

ERRGW3540EF0C001

Message insertion

04/24/1998

Message insertion time

19:59:08

FIG. 20

Event Search

Start Search

Cancel

Search Controls

Date

Time

Server

Host Name

User Name

Service Name

Event ID

Routine

Event Number

Message Search String

21/28

FIG. 21

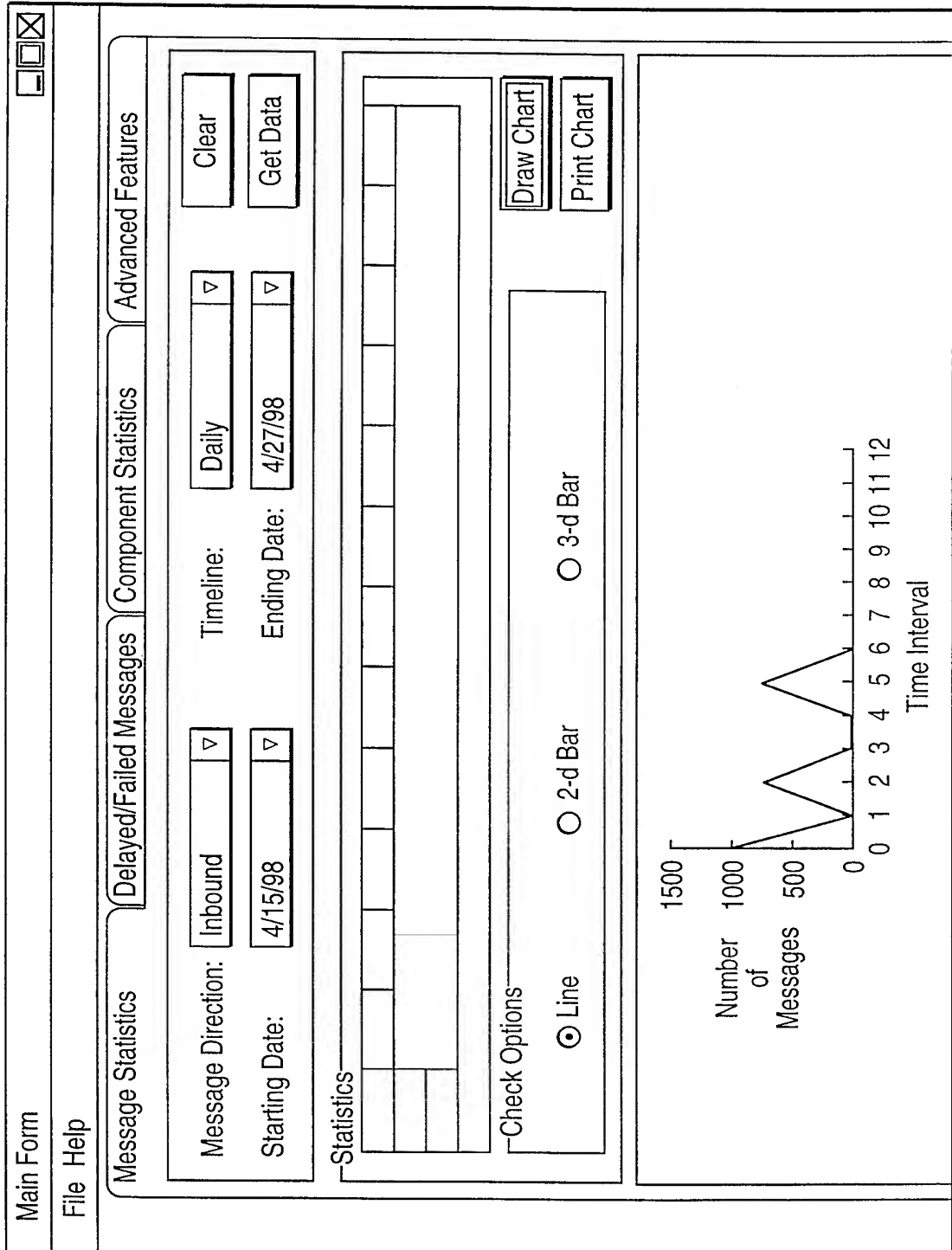


FIG. 22

Main Form

File Help

Message Statistics

Delayed/Failed Messages

Component Statistics

Advanced Features

Starting Date: 4/15/98

Ending Date: 4/27/98

Get Data

Failed Messages By Application

| Application Name | Total |
|------------------|-------|
| CSW | 1 |
| GWR | 4 |
| GXR | 4 |

Failed Messages By Reason

| Failed Reason | Total |
|-----------------------------|-------|
| 28 - TEST LINE FORMAT ERROR | 1 |
| SWIFT Reject | 4 |
| Wrong Route | 4 |

Message Delay

Longest Message Delay

2 days 19:39:19

% Message Delay

05:44:13

23/28

FIG. 23

Main Form

File Help

Message Statistics

Delayed/Failed Messages

Component Statistics

Advanced Features

Message Direction:

Inbound

Starting Date:

4/15/98

Ending Date:

4/19/98

Grouping Criteria

Message Type:

Message Service:

Message Description:

Close

Get Statistics

Statistics

| | | | | | | | |
|-----------------------|---|----|-----|-----|-----|----|--|
| Swift BIC Server Type | | | | | | | |
| Number Of Messages | 6 | 81 | 172 | 173 | 312 | 55 | |

FIG. 24

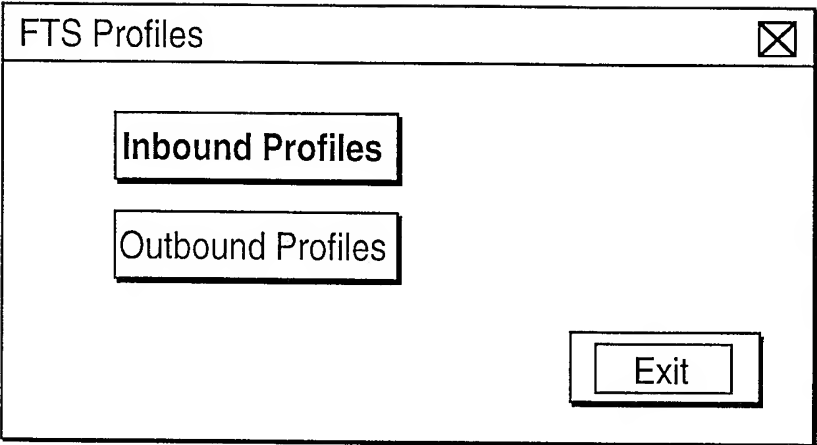
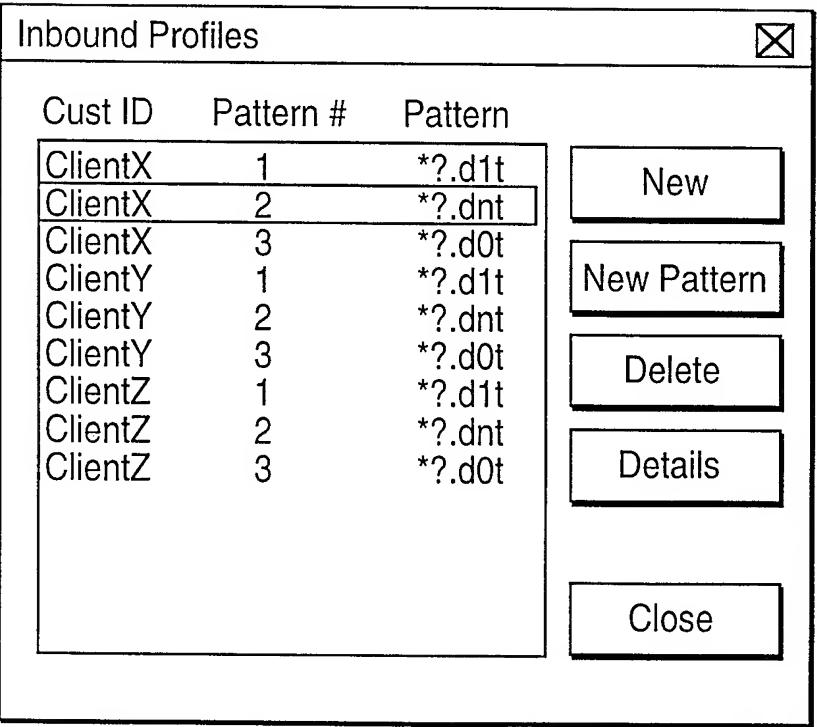
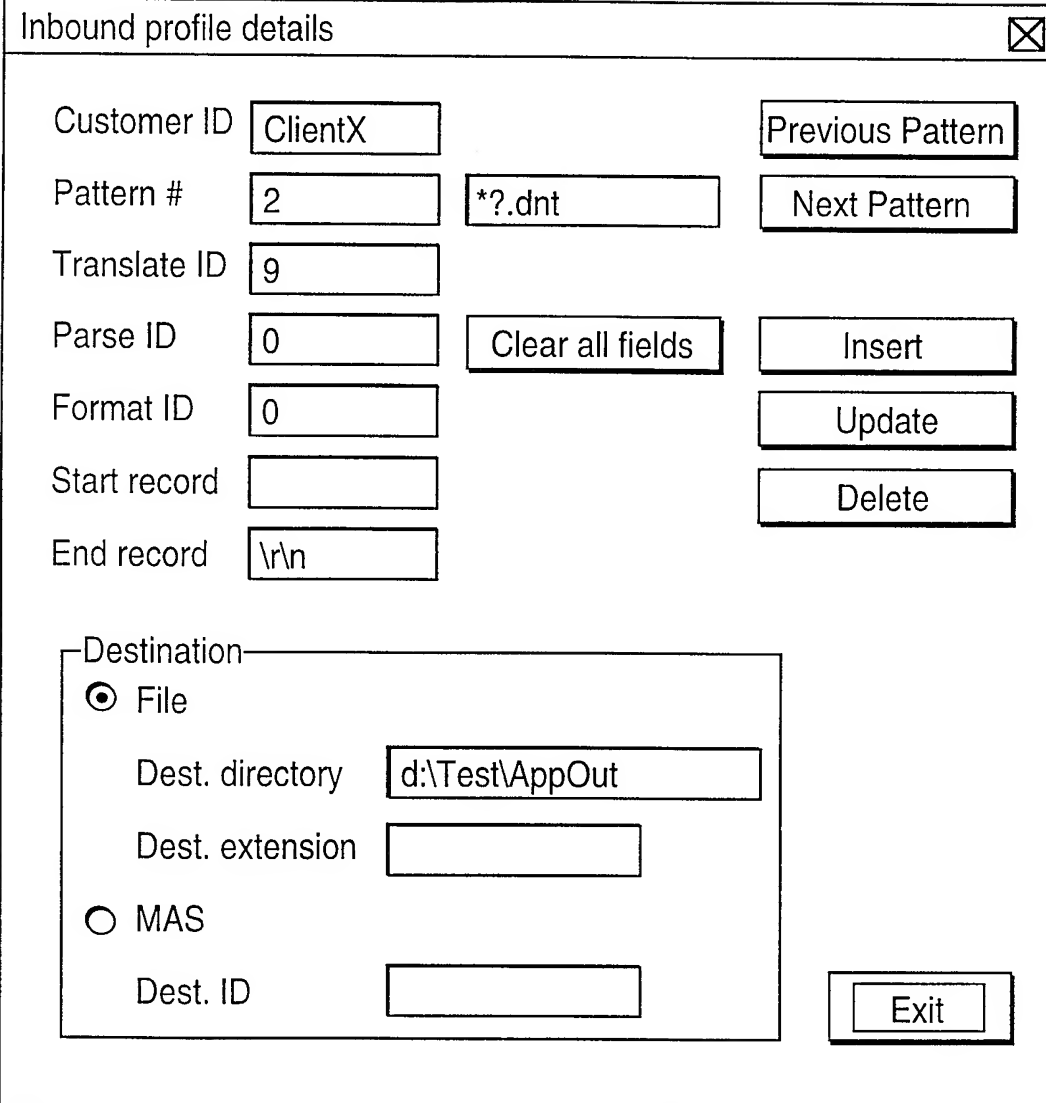


FIG. 25



25/28

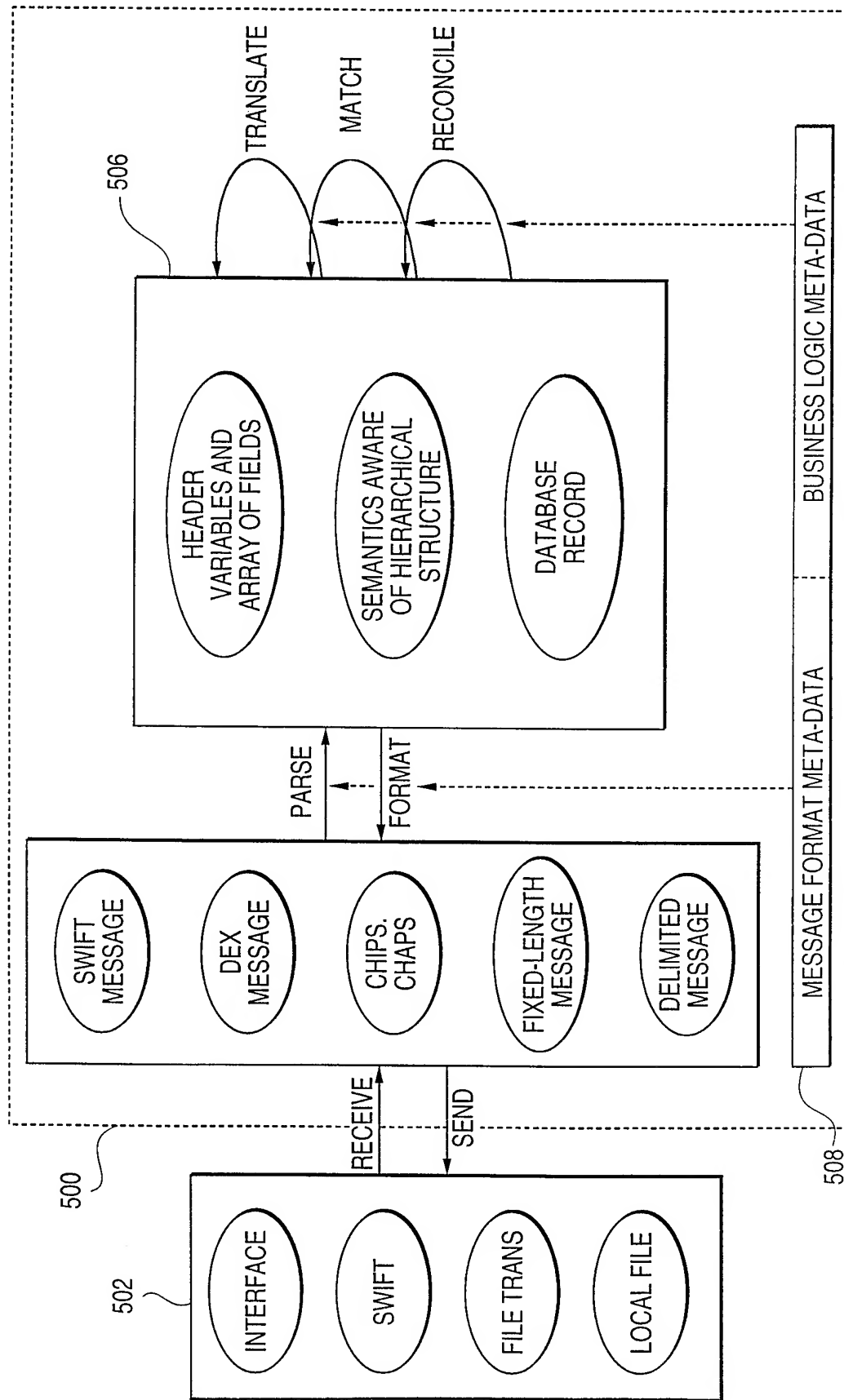
FIG. 26

The dialog box titled "Inbound profile details" contains the following elements:

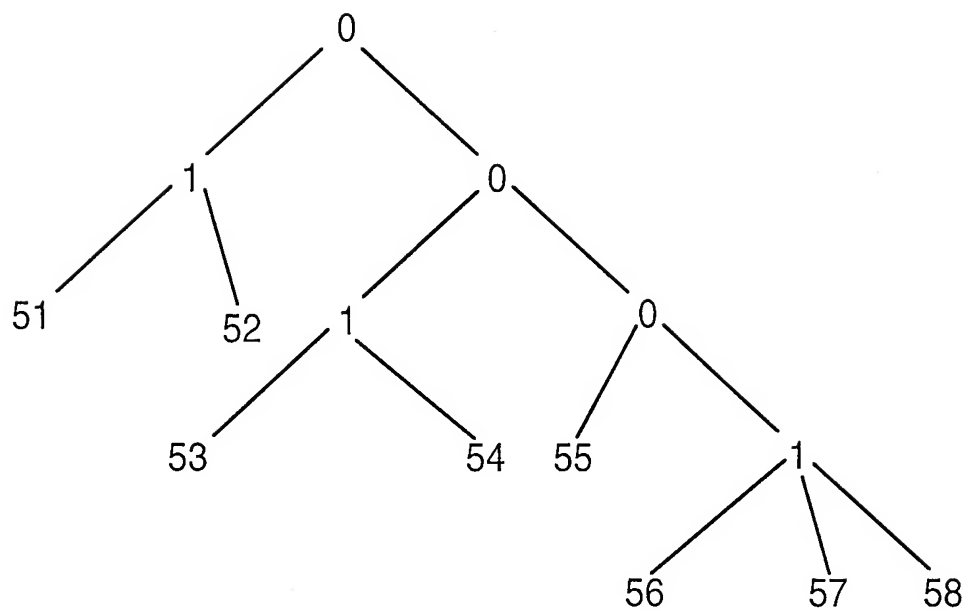
- Customer ID:** Text box with "ClientX".
- Pattern #:** Text box with "2".
- Translate ID:** Text box with "9".
- Parse ID:** Text box with "0".
- Format ID:** Text box with "0".
- Start record:** Empty text box.
- End record:** Text box with "\r\n".
- Buttons:** "Previous Pattern", "Next Pattern", "Clear all fields", "Insert", "Update", "Delete", and "Exit".
- Destination Section:**
 - File (selected):**
 - Dest. directory:** Text box with "d:\Test\AppOut".
 - Dest. extension:** Empty text box.
 - MAS:**
 - Dest. ID:** Empty text box.

26/28

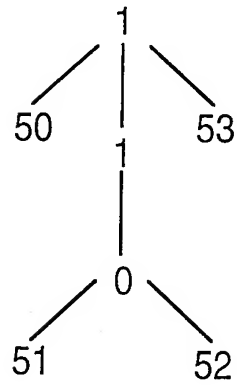
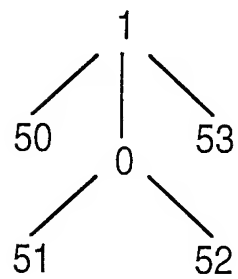
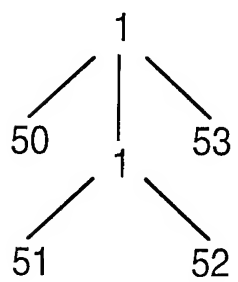
FIG. 27



27/28

FIG. 28

28/28

FIG. 29**FIG. 30****FIG. 31**

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/10930

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : H01J 13/00

US CL : 705/35

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 705/35

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS

C. DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|-----------|--|-----------------------|
| Y | Stallings, "Data and Computer Communications," Fourth Edition, Macmillan Publishing Company, 1994, pp. 170-171, 133-138. | 13, 15-42 |
| X __,P | US 5,424,938 A (WAGNER et al.) 13 JUNE 1995, figures, abstract, claims, col. 4. | 1-12, 14 13, 15-42 |
| Y | | |
| A,P | US 5,710,889 A (CLARK et al.) 20 JANUARY 1998, figures, abstract, claims | all |
| A,E | US 5,787,402 A (POTTER et al.) 28 JULY 1998, figures, abstract, claims | all |

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

| | |
|---|--|
| * Special categories of cited documents: | *T* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention |
| *A* document defining the general state of the art which is not considered to be of particular relevance | *X* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone |
| *E* earlier document published on or after the international filing date | *Y* document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art |
| *L* document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified) | *&* document member of the same patent family |
| *O* document referring to an oral disclosure, use, exhibition or other means | |
| *P* document published prior to the international filing date but later than the priority date claimed | |

Date of the actual completion of the international search

11 AUGUST 1998

Date of mailing of the international search report

20 OCT 1998

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703) 305-3230

Authorized officer

DANIEL PATRU

Telephone No. (703) 305-9605